Conception et configuration de réseaux TSN guidées par les modèles

Maxime Samson

École doctorale IAEM Lorraine

Conception et configuration de réseaux TSN guidées par les modèles

THÈSE

présentée et soutenue publiquement le 24 janvier 2024

pour l'obtention du

Doctorat de l'Université de Lorraine

(mention informatique)

par

Maxine Samson

Composition du jury

Président :	Dr. Liliana Cucu-Grosjean	Directeur de recherche, INRIA
Rapporteurs :	Pr. Katia Jaffres-Runser	Professeur, IRIT-INPT/ENSEEIHT
	Dr. Marc Boyer	Directeur de recherche, ONERA
Examinateur :	Pr. Emmanuel Grolleau	Professeur, ISAE-ENSMA, LIAS
Directeur de thèse :	Pr. Ye-Qiong Song	Professeur, Loria
Encadrants :	Dr. Éric Dujardin	Ingénieur de recherche, Thales
	Dr. Thomas Vergnaud	Ingénieur de recherche, Thales
Invité :	Dr. Laurent Ciarletta	Maître de conférence, Loria

Laboratoire Lorrain de Recherche en Informatique et ses Applications — UMR 7503

Mis en page avec la classe thesul.

Remerciements

Je tiens à exprimer ma gratitude envers toutes les personnes qui m'ont permis, directement ou indirectement, de mener ce travail de thèse à bien.

Je remercie donc l'ensemble de mes encadrants, à commencer par Ye-Qiong Song, pour avoir dirigé cette thèse, pour m'avoir supporté, dans tous les sens du terme, durant toute la durée de cette thèse et pour avoir toujours cru en moi.

J'adresse également mes remerciements à Thomas Vergnaud pour la grande richesse de nos échanges et pour son soutien indéfectible dans les bons moments comme dans les plus difficiles sans lequel je n'aurai pas pu mener ce travail à son terme.

Enfin je remercie Éric Dujardin pour son encadrement, pour la grande expérience qu'il a su partager et pour la pertinence de ses conseils.

Je remercie mes collègues du laboratoire LISL de Thales Research & Technology et de l'équipe SIMBIOT du Loria pour les moments passés ensemble et pour leur partage de connaissances et d'expériences.

Je remercie également les membres de mon jury, Liliana Cucu-Grosjean, Katia Jaffres-Runser, Marc Boyer et Emmanuel Grolleau, pour le temps consacré à la lecture de ce manuscrit ainsi qu'à la soutenance, mais également pour leurs remarques, conseils et questions très pertinentes.

Pour finir, je veux remercier toutes les personnes qui m'ont soutenu, hors du cadre de mes travaux, tout au long de ces années passées en thèse.

Merci à ma famille pour leur amour inconditionnel.

Merci à Éléonore pour son amitié, ses encouragements, et pour l'aide la plus précieuse qu'on ne m'ait jamais apportée.

Merci à Marie, Tristan-Pablo, Vincent et Léo pour avoir partagé avec moi des instants inestimables, et pour m'avoir aidé à traverser les périodes de renoncement.

Enfin, merci aux membres du serveur « Metal Leaping Glass », et plus particulièrement à Marc, Max, Carlos, Jason, Ivo et Dionysos pour leur humour, leur utilisation si particulière d'un vocabulaire toujours plus étrange et pour avoir réussi à me faire oublier mon travail le temps de quelques soirées.

À mon père

Table des matières

Liste des figures

Liste des tableaux

Liste des listings

Liste des publications

M. Samson, Th. Vergnaud. Automatic Generation of Test Oracles from Component Based Software Architectures. *Testing Software and Systems*, pages 261–269. Springer International Publishing, 2019.

M. Samson, Th. Vergnaud, É. Dujardin, L. Ciarletta, and Y.-Q. Song. Une Approche de Génération Automatique de Configuration Basée sur les Modèles pour les Réseaux TSN. *L'École d'Été Temps Réel (ETR)*, Poitiers, France, 2021.

M. Samson, Th. Vergnaud, É. Dujardin, L. Ciarletta, and Y.-Q. Song. A Model-Based Approach to Automatic Generation of TSN Network Simulations. *IEEE 18th International Conference on Factory Communication Systems (WFCS)*, Pavia, Italie, 2022.

M. Samson, Th. Vergnaud, É. Dujardin, L. Ciarletta, and Y.-Q. Song. Computing Data Streams in Real-Time Networks from Component-Based Software Engineering. *IEEE 28th International Conference on Emerging Technologies and Factory Automation (ETFA)*, Sinaia, Roumanie, 2023.

Introduction générale

Ce manuscrit présente une thèse CIFRE réalisée conjointement au Loria, le Laboratoire lorrain de recherche en informatique et ses applications, de l'Université de Lorraine, et à Thales Research & Technology. Cette thèse a été accueillie au sein de l'équipe SIMBIOT du Loria. L'équipe SIMBIOT est spécialisée dans la conception et la validation des systèmes cyber-physiques, elle s'intéresse aux capacités de calcul et de communication de ces systèmes. Thales Research & Technology est le centre de R&T français du groupe Thales, dont les secteurs d'activités sont : l'aéronautique, le transport terrestre, la défense et la sécurité des systèmes d'information. La thèse a été accueillie au sein du laboratoire Ingénierie Système et Logicielle (LISL). Ce laboratoire est spécialisé en recherche appliquée dans les domaines de la modélisation et de la conception de systèmes, de logiciels et de réseaux embarqués. Cette thèse a été réalisée entre janvier 2020 et octobre 2023.

L'évolution des systèmes embarqués mène à un besoin grandissant en bande passante ainsi qu'à la nécessité pour les réseaux qu'ils utilisent de supporter différents types de trafic, de différents niveaux d'importance. Ces systèmes sont particulièrement utilisés dans des domaines comme l'automobile, l'aviation, l'automatisation industrielle et le spatial. Un exemple simple d'un cas d'application est une commande de freinage : ces systèmes doivent apporter la garantie que le temps de réponse entre la demande de freinage (par pression d'une pédale de frein par exemple) et l'activation des freins soit à la fois borné et très court. Cette garantie doit être maintenue à tout instant, même lorsque le réseau permet aussi la circulation de flux de données moins importants, comme des flux audio à destination d'enceintes. Lorsque l'on conçoit un réseau devant prendre en charge différents types de trafics, à la fois critique et non critique, il y a plusieurs possibilités :

— ne rien faire face au délai du trafic critique causé par du trafic non critique, cette solution n'est pas viable ;

— grandement surdimensionner les capacités du réseau, cette solution est très coûteuse ;

— utiliser différents réseaux séparés pour les différents types de trafic, cette solution est également très coûteuse ;

— utiliser une technologie de réseaux temps réel offrant une bande passante importante et permettant de garantir que le trafic non critique ne peut pas gêner le trafic critique et que ce dernier respectera toujours ses échéances.

Les travaux présentés dans cette thèse se concentrent sur les réseaux TSN (*Time-Sensitive Networking*), qui correspondent à la dernière solution citée ci-dessus. Le contexte de ces travaux est également celui des systèmes embarqués, qui repose sur des communications de type contrôle-commande et de type supervision (comme des flux multimédia).

Nous étudions le processus de conception et de configuration de réseaux temps réel utilisant les standards TSN. Le groupe de travail IEEE 802.1 TSN a publié un ensemble de standards

ajoutant de nouvelles fonctionnalités aux normes utilisés par les réseaux Ethernet commutés. L'objectif de ces nouvelles fonctionnalités est de permettre la mise en place de réseaux Ethernet déterministes, ce qui rend possible leur utilisation dans le cadre d'applications temps réels.

L'obtention de cette propriété de déterminisme en présence de trafic à la fois critique et non critique a néanmoins un coût : l'augmentation de la complexité dans la conception et dans la configuration de ces réseaux. Ces nouvelles fonctionnalités entraînent une augmentation de l'effort de configuration destiné à déployer un réseau capable de respecter ses exigences temps réel. De plus, le processus de conception de ces réseaux faisant usage d'outils de conception tel que des simulateurs réseau, cette augmentation de complexité se répercute également sur ces derniers.

Parmi les nombreux standards, deux mécanismes sont particulièrement intéressants et ont suscité beaucoup d'études, il sagit de IEEE 802.1Qbv, qui définit le *Time Aware Shaper* (TAS), et IEEE 802.1Qav, qui définit le *Credit Based Shaper* (CBS). TAS permet l'allocation de fenêtres temporelles aux flux de données aux moments exacts de leur besoin de transmission, au niveau de chaque port de sortie d'un nœud du réseau. CBS permet l'allocation de la proportion de bande passante nécessaire à des flux de données de niveaux de priorité différents afin de garantir leur échance, tout en évitant le problème de famine des flux de données de niveaux de priorité inférieurs.

Dans cette thèse, nous apportons une réponse à deux problématiques :

— La configuration du TAS nécessite un couplage fort entre une application, les flux de données qu'elle génère et la configuration des mécanismes d'ordonnancement (TAS et CBS) dans un système synchronisé (par gPTP définit dans IEEE802.1AS). Les outils de conception et de simulation classiques font l'hypothèse que les flux de données et leurs contraintes temps réel sont parfaitement spécifiées en amont et ils sont utilisés pour vérifier ensuite le respect ou non de ces contraintes. Dans la pratique, il est rare qu'une application avec des tâches distribuées soit aussi finement spécifiée, notamment en ce qui concerne la période d'exécution et la phase de démarrage des tâches. De plus, lors de la phase de la conception, suite à une configuration invalidée par la simulation, ou suite à une modification de l'application (e.g., ajout d'un nouveau nœud au réseau), il est souvent désirable d'apporter des changements dans la spécification de l'application elle-même. Il devient donc nécessaire d'assurer la cohérence entre une configuration réseau et l'application qui utilisera ce réseau. C'est un défi à la fois méthodologique et d'ingénierie.

— Les méthodes et outils de génération automatique de configuration pour le TAS existent, mais ceux qui permettent la configuration du TAS et du CBS lorsqu'ils sont utilisés de façon conjointe se font rares et soulèvent de nouveaux sujets de recherche. Notamment l'allocation de la bande passante nécessaire pour les flux de données ordonnancés par le CBS, qui doit prendre en compte l'impact des fenêtres temporelles du TAS sur la disponibilité des ressources du réseau.

Nous proposons une approche outillée d'assistance à la conception de réseau TSN qui repose sur la modélisation du réseau, des applications qui l'utiliseront et la génération automatique (notamment de modèles de simulation). Nous commençons par une approche de modélisation pour les réseaux TSN, que nous lions ensuite à une approche de modélisation logicielle, basée sur les principes du *Component-Based Software Engineering*, afin de mettre en place un processus de génération automatique qui complète le modèle du réseau avec les flux de données. À l'aide de ce modèle, nous développons une méthode de calcul de la configuration de deux mécanismes d'ordonnancement de TSN : le *Credit-Based Shaper* et le *Time Aware Shaper*. Enfin, nous avons développé un outil générant un ensemble de modèles de simulation à destination de plusieurs

simulateurs réseau ainsi que les fichiers de configuration des équipements réseau.

Notre approche prend en charge la conception de l'application (la partie logicielle) et du réseau. Cela nous permet de lier deux approches de modélisation ce qui nous permet d'assurer la cohérence entre les besoins des applications et la configuration du réseau, tel qu'il sera expliqué par la suite.

Ce manuscrit est organisé en deux parties. Dans la première, nous présentons le contexte dans lequel s'inscrivent les travaux de cette thèse : les réseaux TSN dans les systèmes embarqués temps réel.

Nous commençons par présenter les concepts du temps réel et plus spécifiquement des réseaux temps réel dans le chapitre 1. Nous présentons également différentes technologies de réseaux temps réel utilisées dans différents domaines d'application tels que l'automobile, l'automatisation industrielle et l'avionique.

Nous présentons ensuite les réseaux TSN et les standards utilisés par ceux-ci dans le chapitre 2. Ces standards définissent plusieurs mécanismes d'ordonnancement du trafic sur lesquels nous nous concentrerons dans le reste de ces travaux.

Enfin, nous présentons les concepts de modélisation que nous utilisons dans la suite de cette thèse dans le chapitre 3. Nous nous concentrons sur les approches de modélisation logicielle déjà existantes. Nous terminons par une présentation des différents outils de conception, comme des simulateurs réseau, qui seront utilisés par la suite.

Dans la deuxième partie nous présentons les différentes contributions de cette thèse. Nous commençons par présenter les problématiques auxquelles les contributions répondent dans le chapitre 4. Nous présentons également l'exemple sur lequel nous nous appuierons dans l'ensemble de ce manuscrit.

La première contribution que nous présentons est notre approche de modélisation réseau, dans le chapitre 5. Ces travaux sont également abordés dans les publications [SVD+21] et [SVD+22]. Nous définissons l'ensemble des concepts de modélisation que nous utilisons pour représenter chaque éléments d'un réseau TSN.

La deuxième contribution que nous présentons est le calcul automatique d'un modèle des flux de données à partir de la modélisation de l'architecture logicielle du système, dans le chapitre 6. Ces travaux sont également présentés dans le publication [SVD+23]. Cette contribution inclut la définition d'un métamodèle du comportement des applications et la méthode de calcul des différents éléments caractérisants un flux de données. Le métamodèle que nous avons définit est également présenté dans [SV19].

La troisième contribution que nous présentons est une méthode de calcul de la configuration d'un mécanisme d'ordonnancement de TSN : le *Credit-Based Shaper*. Cette contribution est présentée dans le chapitre 7. Cette méthode de calcul permet d'obtenir la proportion minimum de bande passante à réserver pour respecter les contraintes temps réel. L'intérêt principal de notre approche est la prise en compte de l'utilisation combinée du *Credit-Based Shaper* et du *Time Aware Shaper*, qui produit une interaction nécessitant une proportion de bande passante plus importante pour les flux de données gérés par le CBS.

Nous présentons ensuite l'outil de génération automatique de modèles de simulation, de fichiers de configuration pour du matériel et de documentation, MoBACT, dans le chapitre 8. Cet outil est également présenté dans les publications [SVD+21] et [SVD+22].

Enfin, nous présentons l'application de notre approche à notre système exemple et à plusieurs de ses évolutions dans le chapitre 9. Nous présentons également l'application de notre approche à un système plus complexe. Nous donnons ensuite une conclusion générale et présentons plusieurs perspectives de travaux futurs.

Première partie

Présentation générale du contexte

Chapitre 1

Les réseaux temps-réels

Dans ce chapitre, nous introduisons les concepts fondamentaux des systèmes et des réseaux temps réel nécessaires à la compréhension de la suite de ces travaux de thèse. Le domaine du temps réel utilise un ensemble de termes spécifiques que nous définirons. Ces travaux se concentrant sur les réseaux temps réel, nous rappellerons ce qu'est le modèle OSI et où nous nous plaçons dans celui-ci. Nous définirons les concepts utilisés afin de décrire les différents éléments d'un réseau et nous aborderons la notion de qualité de service. Enfin, nous introduirons les différentes technologies existantes permettant la mise en place de réseaux temps réel.

1.1 Définition du temps-réel

Dans le cadre de ces travaux, nous nous intéressons à la conception des réseaux temps réel utilisés dans les systèmes embarqués. Les systèmes embarqués désignent des ensembles de composants – capteurs, calculateurs et actionneurs – regroupés dans des machines. Ils sont utilisés dans de nombreux secteurs différents : l'automobile, l'avionique, l'automatisation industrielle.

Ces systèmes embarqués sont généralement soumis à un ensemble de contraintes auquel échappent le reste des systèmes informatiques. Ces contraintes concernent la taille, le poids et la consommation énergétique des systèmes embarqués, connus sous l'appellation SWaP (*Size Weight and Power*).

Étant utilisés dans des domaines dans lesquels la sécurité des utilisateurs est souvent primordiale, les systèmes embarqués ont une forte responsabilité qui se traduit par la nécessité d'une correction logique et temporelle. La correction logique implique que le système se comporte exactement de la façon dont il a été conçu et ce indépendamment des conditions : pour un état initial donné, le comportement du système doit toujours être identique (cette propriété est appelée déterminisme). La correction temporelle implique, comme il sera détaillé plus loin, que ces systèmes apportent la garantie que leur temps de réponse ne dépasse jamais une certaine borne.

La correction temporelle est en réalité confondue avec la notion de temps réel. Un système temps réel est un système qui est contraint de respecter une borne temporelle supérieure, aussi appelée échéance, sur son temps d'exécution ou de transmission de données. Par exemple, des commandes de vol électriques à bord d'un avion nécessitent ce type de garantie avec une échéance très courte ; on considèrerai que ce système ne fonctionne pas si son temps de réponse était supérieur à 1 minutes, alors que ce délai ne poserais pas de problème dans le cas du téléchargement d'un fichier sur internet. Une échéance doit donc être non seulement finie mais également adaptée aux exigences du système auquel elle s'applique.

La problématique du temps réel n'est pas uniquement liée aux systèmes embarqués, elle est présente dans de nombreux domaines dont l'ordre de grandeur des échéances peut aller de la microseconde, dans le cas de systèmes de radars, à l'heure, dans le cas de commande de procédés chimiques. On peut distinguer plusieurs types de contraintes temps réel :

— Temps réel souple : le non respect de l'échéance entraîne une perte de qualité progressive, comme pour la latence dans un jeu vidéo en ligne ;

— Temps réel dur : l'échéance doit être respectée dans tous les cas, sans quoi le système est considéré comme défaillant ;

— Temps réel ferme : le non respect de l'échéance entraîne la perte totale de la valeur du système sans pour autant le rendre défaillant, comme pour la transmission de vidéo, où une image perd sa valeur dès que l'image suivante est diffusée mais où la perte de quelques images reste acceptable.

Le respect de l'échéance entraîne la définition de la notion de criticité, qui désigne la forme de responsabilité qu'on associe à une tâche ou à une communication du système. La criticité peut être déclinée en plusieurs niveaux différents en fonction de la qualité de l'échéance à respecter. Des systèmes dans lesquels cohabitent plusieurs niveaux de criticité différents sont appelés systèmes à criticité mixte. Ces systèmes sont soumis à une problématique supplémentaire : il faut empêcher la partie à faible criticité du système d'impacter négativement la partie à haute criticité, la partie critique du système.

1.2 Le modèle OSI

Le modèle OSI (Open System Interconnection model) [OSI94] est une norme des communications réseau des systèmes informatiques. C'est un modèle conceptuel permettant de décrire les différentes fonctions mises en œuvre lors de communications prenant place dans un réseau informatique. Bien qu'il ne soit plus utilisé tel quel, ce modèle permet de définir des concepts de base couramment utilisés dans ce domaine.

Ce modèle définir sept couches différentes numérotées de 1 à 7 allant du plus bas niveau, la couche physique, au plus haut, la couche applicative finale. Les couches pertinentes dans le cadre de nos travaux sont les quatre couches de plus bas niveau.

La figure 1.1 présente l'ensemble des couches du modèle OSI ainsi que le nom donné aux échanges réalisés par les quatre premières couches.

Médium de communication

Le médium de communication, parfois appelé couche zéro, est le moyen de transmission du signal physique entre l'émetteur et le destinataire. Des exemples de médium de communications sont les câbles en cuivres, la fibre optique ou les ondes radio.

Couche physique

La couche physique désigne le protocole utilisé pour transmettre les informations dans un format binaire sur le médium de communication de façon analogique ou numérique, comme le codage Manchester présenté dans le chapitre 6 du livre [Puj08].

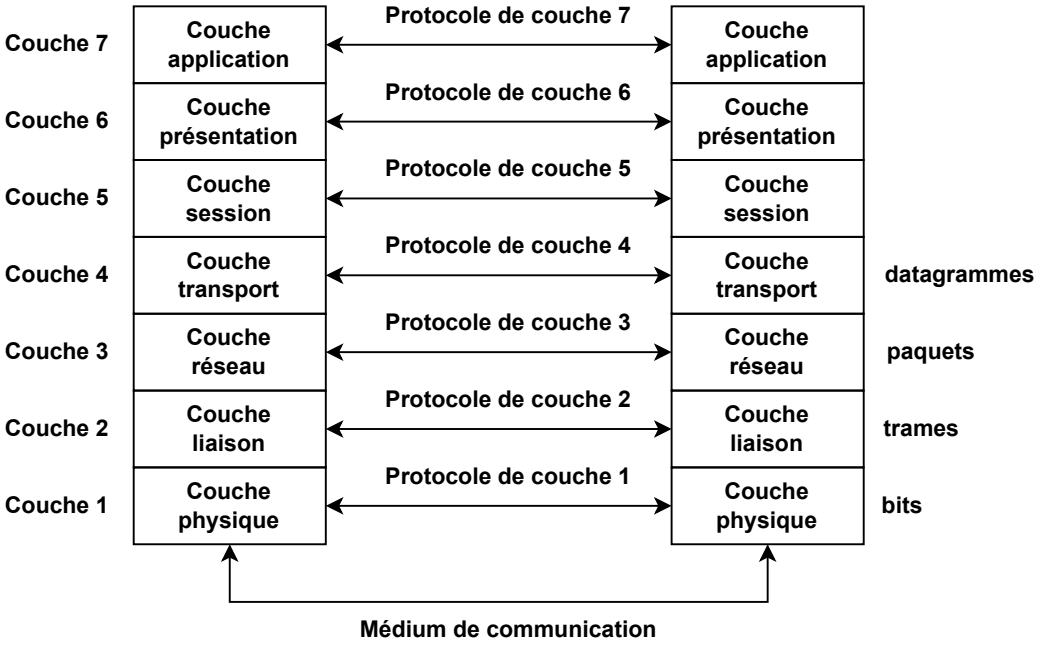


FIGURE 1.1 – Schéma du modèle OSI.

Couche lisaison

La couches liaison est en charge de la gestion des communications entre deux machines directement reliées ou reliées par le biais d'un ou plusieurs équipements réseau, des commutateurs. Cette couche regroupe les données au format binaire en trames, une entité transmise d'un bloc sur le réseau. Le protocole le plus connus de cette couche est Ethernet et sa sous couche MAC (*Media Access Control*) [MAC18].

Couche réseau

La couche réseau permet de déterminer le parcours des données au cours de son chemin dans le réseau entre sa source et sa destination, appelé routage. Les paquets contiennent les informations nécessaires pour les guider vers leur destination, comme l'adresse de cette dernière. Le protocole le plus utilisé pour cette couche est IP (*Internet Protocol*) [IP81].

Couche transport

La couche transport introduit la notion de port et permet la transmission de données de bout en bout entre des processus déployés sur des systèmes différents, indépendamment des couches inférieures. Les protocoles les plus représentés pour cette couche sont TCP (*Transport Control Portocol*) [TCP81] et UDP (*User Datagram Protocol*) [UDP80], dont les messages sont appelés des datagrammes.

Dans la suite de ces travaux, nous nous concentrerons sur la couche liaison et nous mentionnerons l'utilisation de protocoles des couches réseau et transport. Nous ne détaillons pas les couches supérieures car elles ne sont pas pertinentes dans le cadre de ces travaux.

1.3 Concepts utilisés dans le domaine des réseaux

Lors de l'étape de conception d'un réseau, il est nécessaire de maîtriser un ensemble de concepts. Nous définissons ici les termes : trame, nœud réseau, terminal, pont, topologie réseau et classe de trafic.

Définition 1.3.1. (Trame). La trame est la structure de base utilisée pour la transmission d'un ensemble de données de taille finie. C'est le terme utilisé pour l'unité de communication de la couche 2 du modèle OSI. Une trame est composée d'un en-tête, des données à transmettre (la charge utile), et d'un postambule. Un paquet IP étant d'une couche supérieure (la couche 3), il sera encapsulé comme charge utile d'une trame. Dans les usages les plus courants d'Ethernet, la charge utile comporte entre 46 et 1500 octets et l'en-tête 18 ou 22 octets (le format d'une trame Ethernet sera présenté en détail dans la section 2.1).

Définition 1.3.2. (Nœud). Un nœud est un composant physique du réseau, dans notre cas il peut désigner un terminal ou un pont, dont les définition sont données ci-après.

Définition 1.3.3. (Terminal [1]). Un terminal est une machine reliée au réseau dont le rôle est d'émettre, de recevoir des données, ou les deux à la fois.

Définition 1.3.4. (Pont [2]). Un pont est un élément du réseau qui permet de relier plusieurs nœuds du réseau entre eux. Les trames transitent par les ponts, arrivant par un port et étant transmises par un autre, jusqu'à leur destination.

Définition 1.3.5. (Topologie réseau). La topologie du réseau désigne l'agencement des nœuds du réseau et la façon dont ils sont connectés les uns avec les autres à l'aide de lien. Une topologie forme un graphe.

Il existe plusieurs types de topologie réseau : les topologies basées sur la diffusion et les topologies commutées. Il est important de noter que nous ne considérons que des réseaux filaires dans le cadre de ces travaux.

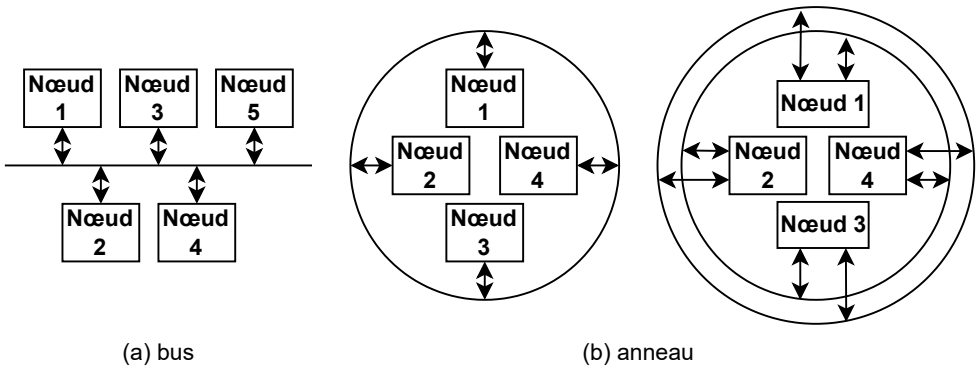


FIGURE 1.2 – Exemples de topologies réseau basées sur la diffusion.

La figure 1.2 présente des exemples de topologies réseau basées sur un mode de diffusion. Ce mode de fonctionnement utilise un unique support de communication, les messages envoyés sur le réseau sont donc reçu par tous les terminaux qui peuvent ensuite décider si le message leur est adressé ou non.

1. Aussi appelé machine ou calculateur et *end-point* ou *end-system* en anglais.
2. Aussi appelé commutateur ou *switch* en anglais.

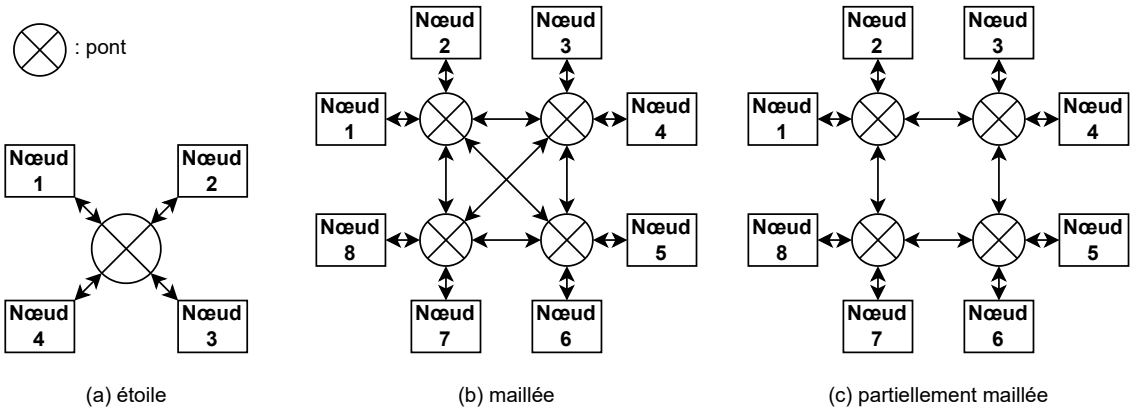


FIGURE 1.3 – Exemples de topologies réseau commutées.

La figure 1.3 présente des exemples de topologies réseau commutées. Ce mode de fonctionnement utilise de nombreux support de communications reliant les nœuds du réseau par paires. Les communications entre les terminaux doivent donc passer par les ponts, chaque passage entre les nœuds le long du chemin empruntés par une communication est appelé un saut.

Dans les réseaux Ethernet modernes, les topologies commutées, utilisant des ponts (ou commutateurs) sont utilisées dans la majorité des cas et c'est donc sur celles-ci que nos travaux se concentreront.

Définition 1.3.6. (Classe de trafic). Une classe de trafic est la classification d'un ensemble de communications réseau entre des terminaux en fonction de leur niveau de criticité.

Il existe des classes de trafic couramment utilisées dans les réseaux temps réel :

— Le trafic au « meilleur effort » [3], cette classe de trafic correspond au trafic qui ne nécessite pas de contraintes temps réel particulières.

— Le trafic critique, cette classe de trafic correspond au trafic dont le niveau de criticité est le plus élevé du système et qui est donc associé aux exigences temps réel les plus dures.

Lors de l'étape de conception du système, il est possible de définir d'autres classes de trafic spécifiquement pour les besoins du système et de leur associer des exigences temps-réel adaptées.

1.4 La qualité de service

La notion de qualité de service est particulièrement importante dans le domaine des systèmes embarqués temps réel car elle englobe l'ensemble des caractéristiques qu'un tel système doit être capable de garantir.

Définition 1.4.1. (Qualité de service). Dans le contexte de communications réseau, la qualité de service désigne l'ensemble des propriétés qu'un réseau temps réel doit être capable de garantir. Ces propriétés incluent la latence de bout en bout, la gigue maximale tolérée et la tolérance aux fautes. Par exemple, une classe de trafic peut nécessiter la garantie que la latence de bout en bout des communications qui lui sont associées ne dépasse jamais 1 ms et que sa gigue ne dépasse jamais 200 µs.

3. Aussi appelé *Best Effort* en anglais.

Une partie de la notion de qualité de service concerne donc l'aspect temporel du comportement d'un réseau temps réel. Nous nous intéressons aux deux concepts suivant dans la suite de ces travaux : la latence et la gigue.

Définition 1.4.2. (Latence). La latence est le temps entre l'émission d'une trame réseau par sa source et la réception de cette trame par sa destination. Cette latence est celle du réseau, elle est la somme des temps de transmission entre les nœuds, de propagation sur les liens, de traitement par les ponts et d'attente due aux mécanismes d'ordonnancement. La latence désigne la latence de bout en bout, sauf indication contraire.

Définition 1.4.3. (Gigue). La gigue est la variation de la latence.

Les systèmes embarqués temps réel fonctionnant par l'échange de communications d'un niveau de criticité élevé, il est nécessaire qu'ils disposent de mécanismes leur permettant de garantir que leurs contraintes de latence et de gigue seront respectées. Les mécanismes les plus importants pour obtenir ces garanties sont les mécanismes d'ordonnancement.

Dans un système dont les ressources de calcul et, plus particulièrement dans le cadre de nos travaux, de communication sont limitées, les mécanismes d'ordonnancement permettent de répartir l'accès à ces ressources entre les différentes communications afin qu'elles puissent toutes respecter leurs contraintes temps réel. Cette nécessité et les problématiques qui en découlent correspondent à la théorie de l'ordonnancement [LL73].

Un algorithme d'ordonnancement est une méthode permettant d'obtenir l'ordonnancement à utiliser. Le cas des systèmes temps réel à criticité mixte rend la création d'algorithmes d'ordonnancement plus complexe, la présence de plusieurs niveaux de criticité ayant un impact sur les mécanismes d'ordonnancement [Ves07]. Les algorithmes d'ordonnancement peuvent également contenir une méthode permettant de déterminer si un ensemble de tâches ou de communications est ordonnançable.

Ces mécanismes permettent de satisfaire les deux contraintes temporelles des systèmes embarqués temps réel : borner la latence et borner la gigue, c'est-à-dire garantir que pour l'ensemble des communications, la latence et la gigue de chaque trame transmise sera inférieure à une valeur spécifiée lors de la conception du système.

L'autre partie de la qualité de service concerne la tolérance aux fautes, c'est-à-dire la capacité à maintenir les communications et le respect de leurs contraintes temps réel malgré une panne dans le système. Tout comme pour la gigue et contrairement à la latence, certains systèmes ne nécessitent pas cette propriété mais elle est néanmoins particulièrement importante lorsqu'un système doit pouvoir garantir la sécurité de ses utilisateurs.

Cette propriété peut être assurée par un ensemble de mécanismes fonctionnant ensemble. Ces mécanismes concernent dans un premier temps la détection d'une faute (par exemple la perte d'une trame), l'isolation d'une faute (par exemple l'élimination d'une trame dont l'échéance est passée afin qu'elle ne perturbe pas le réseau) et la récupération d'une faute (par exemple par la retransmission d'une trame perdue).

Dans un second temps, des mesures peuvent être mises en place pour permettre de supporter physiquement une faute : des mesures de redondances. On peut identifier trois stratégies de redondances différentes :

— la redondance froide : les systèmes redondants sont désactivés et ne deviennent actifs que lorsqu'une panne est détectée ;

— la redondance tiède : les systèmes redondants sont activés mais sont en attente (ils ne participent pas aux communications) jusqu'à la détection d'une panne ;

— la redondance chaude : les systèmes redondants sont actifs et participent aux communications (par exemple en transmettant des copies des communications en cours).

1.5 Les technologies de réseaux temps réel

Plusieurs technologies de réseaux temps réels existent et sont utilisées, pour certains, depuis quelques dizaines d'années dans des secteurs différents.

1.5.1 Réseaux automobiles – Controller Area Network (CAN)

Le bus de données CAN [CAN15] (*Controller Area Network*) est un bus de communication, comme présenté dans la figure 1.2, qui est particulièrement répandu dans le domaine automobile. Les bus CAN permettent la mise en place de réseaux déterministes, pouvant donc être utilisés pour garantir des exigences temps réel, pouvant raccorder un grand nombre de terminaux à un même câble plutôt qu'à un ensemble de ponts comme c'est le cas pour les réseaux commutés. L'utilisation d'une topologie bus permet donc d'éviter l'emploi de lignes dédiées pour relier chaque terminal tout en faisant l'économie de poids et d'espace que représente l'absence de ponts, ce qui en fait un candidat intéressant pour les réseaux embarqués dans l'automobile.

Le caractère déterministe des bus CAN est assuré par un principe de résolution des collisions appelé « accès multiple avec écoute de la porteuse et résolution de collisions » (*Carrier Sense Multiple Access / Collision Resolution (CSMA/CR)*. Ce principe implique que chaque terminal « écoute » le médium de transmission avant d'émettre et que les éventuels conflits sont résolus par un système de priorité. Chaque trame est préfixée par un identifiant (d'une taille de 11 ou 29 bits suivant la version utilisée). Lorsqu'une collision survient – c'est-à-dire que deux trames sont émises au même moment par deux terminaux différents – c'est cet identifiant qui permet de sélectionner la trame la plus prioritaire qui pourra poursuivre son émission. Ce principe revient à un ordonnancement à priorité fixe et sans préemption.

La difficulté de configuration de l'ordonnancent pour cette technologie est donc la sélection du niveau de priorité à attribuer à chaque flux de données afin que ses exigences temps réel puissent être respectées. Un avantage important de ce type d'ordonnancement est que la présence d'un contrôleur de bus n'est pas nécessaire.

La limitation principales des bus CAN vient de la bande passante qu'il est possible d'utiliser. La taille d'une trame ne peut dépasser 128 bits – 8 octets – dont seulement la moitié est réservée à la charge utile. La version de CAN offrant la bande passante la plus importante est alors limitée à 1 Mbit/s. De plus, cette bande passante est également limitée par la longueur du bus. Ainsi, une bande passante d'1 Mbit/s ne peut être maintenue que pour un bus d'une longueur inférieure à 40 m et n'est plus que de 10 kbit/s pour une longueur de 5 km.

Une extension de CAN a été proposée pour résoudre ce problème de manque de bande passante : *CAN with Flexible Data-Rate* (CAN FD) [CAN11]. Cette norme plus récente permet d'atteindre une bande passante de 12 Mbit/s.

D'autres travaux existent autour de CAN, ayant notamment pour but de faciliter son passage à l'échelle. C'est le cas de l'extension *Time-Triggered CAN* (TTCAN) [FMHH01] qui permet de synchroniser les terminaux du réseau grâce à la présence d'un nœud maître et de définir des fenêtres temporelles pour isoler la transmission de certains flux de données et ainsi éviter les collisions. La synchronisation a néanmoins un coût car elle monopolise une partie de la bande passante.

1.5.2 Réseaux avioniques – ARINC 664 (AFDX)

Dans le domaine de l'avionique, et plus particulièrement de l'avionique civile, le standard ARINC 429 [ARI01] a été une des premières réponses aux besoins de communications temps réel. Ce standard définit un bus de données unidirectionnel et *multicast* (le nombre de destinataires pour une transmission peut aller jusqu'à vingt). Une trame est composée de 32 bits et la bande passante disponible est soit de 100 kbit/s soit de 12,5 kbit/s. En plus d'une faible bande passante, la mise en place de ces réseaux nécessite que chaque nœud destinataire soit relié au nœud émetteur par un câble ce qui entraîne un poids important de l'infrastructure.

L'augmentation de la complexité des systèmes avioniques a rendu nécessaire la mise en place de réseaux ayant une bande passante plus importante et un câblage moins volumineux.

C'est dans ce but qu'on été développé les réseaux *Avionics Full-Duplex Switched Ethernet* (AFDX), conçus par Airbus, via le standard ARINC664p7 [ARI09]. Ces réseaux sont commutés, ce qui réduit le volume de câble en faisant usage de ponts. Ils sont *full-duplex*, c'est-à-dire qu'ils permettent les communications bidirectionnelles en simultané, et non plus unidirectionnelles. Enfin, ils sont également déterministes et redondants, chaque donnée étant dupliquée et émise sur deux chemins distincts. Ce standard est aujourd'hui le standard de référence dans l'avionique civile.

Les nœuds des réseaux AFDX sont disposés dans une topologie en étoile autour de chaque pont. Pour des raisons de redondance, chaque terminal est relié à deux ponts, chaque pont est dupliquée et chaque trame est émise deux fois.

Les communications prennent place dans des liens virtuels [4], définissant la source, les destinataires et les chemins entre eux. La garantie du respect des exigences temps réel s'appuie sur un mécanisme de réservation de bande passante pour chaque lien virtuel. Les communications ont donc l'assurance de disposer de suffisamment de ressources pour leur transmission. De plus, les liens virtuels sont caractérisés par l'intervalle entre deux transmissions.

La réservation de bande passante couplée avec l'intervalle entre deux transmissions permet de borner le temps de latence de chaque communication. L'ordonnancement, tout comme le routage, est statique et est assuré par un système de priorité attribué à chaque lien virtuel.

Malgré ces propriétés, ces réseaux ne sont pas utilisés pour les parties les plus critiques du système car l'analyse de son ordonnançabilité et l'assignation des niveaux de priorités aux différents flux de données est un problème complexe [FFG06]. Enfin, les réseaux AFDX ne permettent pas la transmission de flux de données de criticité mixte, le trafic « meilleur effort », ne nécessitant pas de respecter des contraintes temps réel, n'est pas supporté et nécessitent donc lui aussi la définition de liens virtuels.

Dans le domaine de l'avionique militaire, c'est le standard MIL-STD-1553B [MIL18] qui est privilégié. Une comparaison détaillée des technologies utilisées dans le domaine de l'avionique est présentée dans [SV08].

1.5.3 Automatisation industrielle – EtherCAT

EtherCAT est un standard qui étend Ethernet, créé spécifiquement pour les capteurs et les actionneurs et donc utilisé dans le domaine de l'automatisation industrielle. Il a été développé par l'entreprise Beckhoff Automation et est standardisé par l'*EtherCAT Technology Group*. Il s'appuie sur les bus de terrains, les deux technologies étant regroupées dans le même standard : [fie23].

4. Appelés *Vitual Links* en anglais.

EtherCAT utilise une topologie en anneau. Une unique trame Ethernet peut être utilisée pour communiquer des informations à plusieurs terminaux différents. Les terminaux extraient les données qui les concernent au fur et à mesure de la lecture de la trame avant de la passer au terminal suivant. Cette même trame peut également être utilisée pour transmettre les informations émises par les différents terminaux.

EtherCAT fonctionne donc grâce à une entité centrale transmettant, potentiellement via une unique trame Ethernet, des informations à l'ensemble du réseau. Il est possible de tirer partie de la grande vitesse de transmission offerte par Ethernet mais EtherCAT n'est pas adapté à d'importants volumes de données comme des flux vidéo.

1.5.4 Systèmes embarqués – TTEthernet

Time Triggered Ethernet (TTEthernet) [SBH+09] est une technologie de réseaux temps réel proposée par la société TTTech. Elle est utilisée de façon générale dans le domaine des systèmes embarqués. Tout comme AFDX (TTEthernet étand le standard ARINC 664), elle permet la mise en place de réseaux Ethernet commutés déterministes.

TTEthernet s'appuie sur la synchronisation temporelle des nœuds du réseau et supporte la criticité mixte avec trois classes de trafic (temps réel dur, temps réel souple ou ferme et « meilleur effort »). Les flux de données sont transmis par les ponts durant des fenêtres temporelles de durée prédéterminée ce qui empêche les classes de trafic moins prioritaires de gêner les classes plus prioritaires.

Un mécanisme de tolérance aux fautes est utilisés pour assurer le maintient des communications en cas de panne d'un pont du réseau.

Les liens d'un réseau TTEthernet fonctionnent en mode *full-duplex* et la bande passante disponible peut aller jusqu'à 1 Gbit/s.

1.5.5 Extension des standards Ethernet par l'IEEE – AVB et TSN

Le standard Ethernet Audio Video Bridging (AVB) [BA21] est le précurseur des standards Time-Sensitive Networking (TSN). Son objectif était de proposer une technologie non propriétaire qui permettent la mise en place de réseaux Ethernet déterministes supportant une criticité mixte en permettant la coexistence de trafic audio, vidéo et « meilleur effort ». Un autre objectif d'AVB était de permettre une réservation dynamique des ressources du réseau afin de supporter l'ajout de nouveaux nœuds au cours de l'utilisation du réseau.

La fonctionnalités principale d'Ethernet AVB est son mécanisme d'ordonnancement : le *Credit-Based Shaper* (CBS) dont le rôle est de lisser la transmission du trafic à la manière d'un saut percé[5]. Cela repose sur le principe de répartition de la bande passante disponible aux différentes classes de trafic de façon à empêcher une classe d'un niveau de priorité supérieur de s'accaparer l'ensemble des ressources. Le fonctionnement du CBS repose tout de même sur un principe de priorité mais assure la répartition de l'accès au réseau par un système de crédit. Chaque classe de trafic est associée à une valeur de crédit qu'elle consomme lorsqu'elle transmet et regagne lorsqu'elle attend pour transmettre. Une valeur de crédit négative empêche la transmission. Ce mécanisme d'ordonnancement est décrit dans le standard IEEE 802.1Qav [Qav16].

Ethernet AVB permet également la synchronisation temporelle des nœuds du réseau grâce au standard IEEE 802.1AS [AS20] et la définition su protocole *Generalized Precision Time Protocol* (gPTP).

5. Appelé *leaky bucket* en anglais.

Enfin, le protocole *Stream Reseration Protocol* (SRP) spécifié dans le standard [Qat10] permet la réservation des ressources du réseau au cours de son utilisation.

Le groupe de travail Ethernet AVB a été renommé *Time-Sensitive Networking* pour les raisons suivantes :

— passer d'un mode de contrôle du réseau décentralise à un mode centralisé ;

— ajouter un mécanisme de préemption de trames permettant l'optimisation des transmissions du trafic appartenant aux classes de trafic critiques ;

— étendre son champ d'application à des domaines au delà de l'audio et de la vidéo.

AVB ne possédait que deux classes de trafic permettant d'obtenir une garantie de respect des exigences temps réel spécifiquement pour du trafic de type audio et vidéo pour lesquels l'accès à la bande passante est le critère prédominant. Afin de pouvoir s'étendre à d'autres domaines, ces standards ont dû proposer un plus grand nombre de classes de trafic ainsi que des mécanismes d'ordonnancement plus adaptés à du trafic ayant des besoins différents.

Ce sont ces raisons qui ont conduit à la création de TSN, qui sera présenté dans le chapitre suivant.

Chapitre 2

Les standards TSN

Comme présenté dans le chapitre 1, plusieurs technologies de réseaux temps réels existent mais la plupart d'entre elles sont spécifiques à certains domaines d'application, chères ou limitées, notamment en termes de bande passante. Dans ce chapitre, nous présentons en détail la technologie sur laquelle porte les travaux de cette thèse : TSN.

Cette technologie prend la forme d'un ensemble de standards ajoutant de nombreuses nouvelles fonctionnalités aux réseaux Ethernet ayant pour but de rendre leur utilisation possible pour la mise en place de réseaux temps réels. Nous commençons donc par présenter ce que sont les réseaux Ethernet commutés.

Nous présentons ensuite les différentes fonctionnalités de TSN qui permettent la conception de réseaux Ethernet déterministes et capables de garantir le respect d'exigences temps réel. Ces nouvelles fonctionnalités portent sur plusieurs aspects du réseau : la synchronisation temporelle, l'ordonnancement du trafic et la tolérance aux pannes.

2.1 Ethernet

Ethernet commuté est une technologie de la couche 2 du modèle OSI, elle est définie par les standards [80222] et [Q18]. Ethernet est extrêmement répandue à travers le monde et est considérée comme la technologie standard des réseaux informatiques domestiques et des infrastructures internet.

Avec Ethernet commuté en mode *full-duplex*, c'est-à-dire que chaque lien peut être utilisé dans les deux sens simultanément, les ports des nœuds ne peuvent être reliés qu'à un unique autre port d'un autre nœud. De ce fait, les collisions de trames sont impossibles. L'accès aux ressources du réseau afin de pouvoir transmettre passe par une file d'attente [6] au niveau du port.

Le standard [Q18] définit le format d'une étiquette à ajouter à l'en-tête d'une trame Ethernet permettant de spécifier le niveau de priorité, *Priority Code Point* (PCP), de la trame. Il y a donc une file par niveau de priorité, au nombre de huit pour TSN.

La bande passante d'un réseau Ethernet commuté peut être comprise entre 1 Mb/s et plusieurs Gb/s. Chaque trame peut contenir une charge utile dont la taille maximale est de 1500 octets (hors trames *jumbo* [jum06]).

Une trame Ethernet est composée de trois parties, comme présenté par la figure 2.1 : un en-tête, la charge utile, et une séquence de vérification de la trame. Cet en-tête contient les adresses MAC de source et de destination de la trame. L'étiquette VLAN spécifiée dans le standard [Q18] est optionnelle, elle permet de renseigner l'identifiant du réseau virtuel utilisé pour

6. FIFO, ou « premier entré, premier sorti »

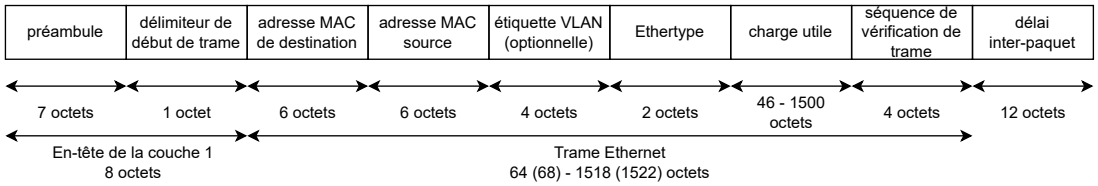


FIGURE 2.1 – Schéma des différentes parties d'une trame Ethernet.

la transmission de la trame et son niveau de priorité. Le champ *Ethertype* porte l'information du protocole utilisé dans la charge utile de la trame. Ce champ contenant la charge utile a une taille minimale de 46 octets car une trame Ethernet ne peut pas avoir une taille totale inférieure à 64 octets.

Á la suite de la charge utile de la trame, un champ contenant une séquence de vérification. Cette séquence est calculée à l'émission en fonction du contenu de la trame et permet, à la réception, de détecter la corruption des données de cette trame. Il est important de noter que cette séquence ne permet pas de corriger d'éventuels erreurs.

Chaque trame Ethernet est suivie d'un délai inter-paquet dont la durée équivaut à la transmission de 12 octets.

Pour résumer, Ethernet est une technologie très connue et très répandue, elle est généralement peur chère et n'est pas propriétaire. Elle offre également une bande passante très importante, nettement supérieure à des technologies comme les bus CAN. Néanmoins, Ethernet commuté ne permet pas la mise en place de réseaux temps réels car elle ne possède pas de moyen de garantir son déterminisme et d'apporter des garanties de respects des exigences temps réel.

2.2 Rendre Ethernet temps-réel

Les standards TSN définissent plusieurs mécanismes d'ordonnancement de trafic, dont certains sont basés sur la division temporelle, qui permettent d'offrir la garantie que les flux de données respecteront leurs exigences temps réel. La synchronisation temporelle, utilisées pour synchroniser les horloges internes des nœuds du réseau, est requise par ces mécanismes, qui peuvent par exemple être basés sur la division temporelle, et est donc une pierre angulaire des fonctionnalités TSN. Ce mécanisme de synchronisation est standardisé dans IEEE 802.1AS [AS20], qui définit le gPTP (*Generalized Precision Time Protocol*).

Pour apporter la garantie que le réseau sera capable de respecter les exigences temps réel des flux de données, les ressources réseaux, la bande passante par exemple, peuvent être réservée au niveau des ports de sortie le long du chemin du flux. Par exemple, la somme des bande passantes requises par chaque flux de données passant par un port de sortie ne doit pas dépasser la capacité de ce port. Le standard IEEE 802.1Qcc [Q18] définit le SRP (*Stream Reservation Protocol*) qui est capable d'effectuer cette réservation.

La figure 2.2 présente les mécanismes d'ordonnancement de trafic présents au niveau des ports de sortie des ponts d'un réseau TSN. Chaque port de sortie possède sa propre politique d'ordonnancement et doit donc être configuré individuellement pour pouvoir respecter les exigences temps réel des flux de données ; cela conduit à une grande quantité de paramètres de configuration et crée une grande complexité dans la configuration d'un réseau TSN.

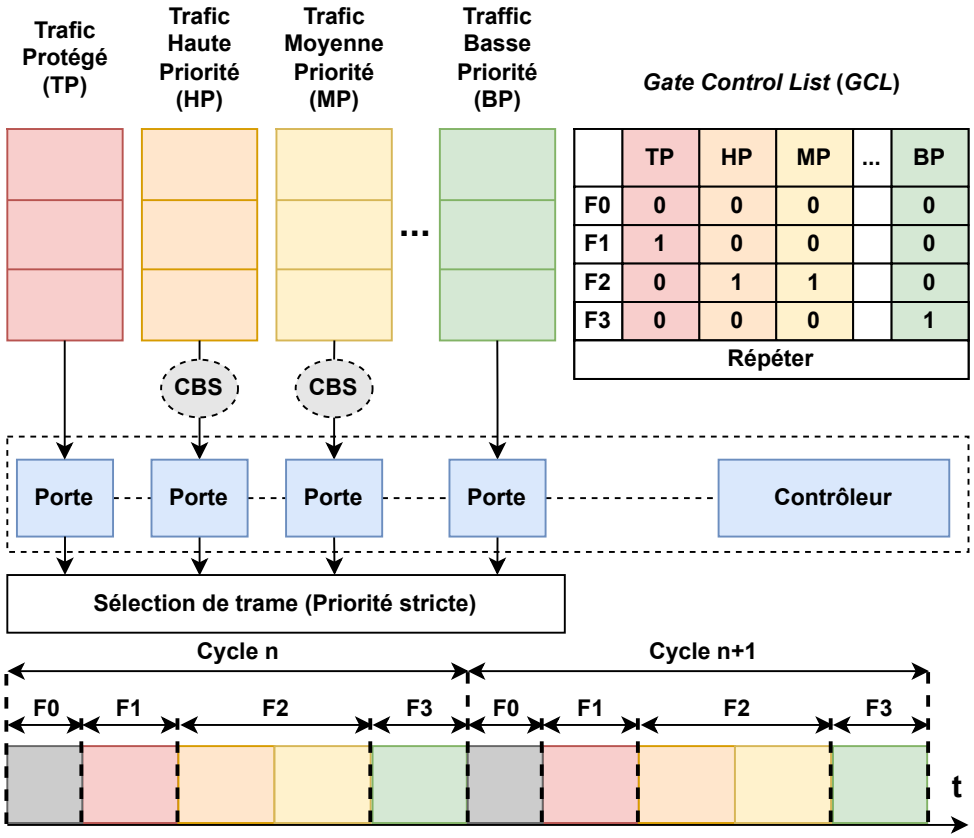


FIGURE 2.2 – Représentation des différents mécanismes d'ordonnancement de trafic au niveau d'un port de sortie TSN.

2.2.1 Synchronisation temporelle

Dans un réseau TSN, afin de pouvoir utiliser des mécanismes d'ordonnancement basés sur la division temporelle, les différents nœuds doivent posséder une notion commune du temps, il doivent être synchronisés. Cette synchronisation permet de coordonner les actions des ponts, comme ordonnancer les transmissions de trames. Un mécanisme d'ordonnancement qui se base sur le temps ne peut fonctionner dans un réseau non synchronisé.

Les réseaux TSN utilisent le protocole défini par IEEE 1588 [15819] : le *Precision Time Protocol* (PTP). Ce protocole utilise un algorithme réparti afin d'élire le nœud du réseau qui servira de référence pour synchroniser l'ensemble des nœuds compatibles avec ce protocole, le *Best Master Clock Algorithm*. Cet algorithme permet de sélectionner le nœud ayant l'horloge interne la plus précise. La précision du protocole PTP est d'un ordre de grandeur inférieur à la microseconde.

La synchronisation des horloges du réseau se fait d'une horloge à l'autre en commençant par les horloges directement reliées à l'horloge élue. Les deux horloges échangent des messages horodatés qui permettent le calcul du décalage entre les deux horloges ainsi que le délai moyen de propagation entre les deux horloges. En répétant ce processus point à point, l'heure de référence est propagée à travers le réseau, tant que les nœuds sont compatibles avec de protocole.

Les standards TSN définissent, dans [AS20], un profile du protocole PTP. Ce profile permet de spécifier quel sous ensemble des fonctionnalités du standard sont souhaitables pour un réseau

TSN, en éliminant le superflu. Le protocole *Generalized Precision Time Protocol* (gPTP) ainsi défini permet également de généraliser ce protocole au réseau Ethernet sans-fil.

2.2.2 *Credit-Based Shaper*

Avoir suffisamment de bande passante à disposition ne permet pas d'apporter la garantie que les exigences temps réel des flux de données seront respectées. Il est également essentiel d'utiliser la bande passante d'une façon qui permet de borner la latence de bout en bout des flux de données critiques sans faire souffrir les autres flux de données d'un phénomène de famine. Le phénomène de famine est ce qui ce produit pour les flux de données non critique lorsque l'ordonnancement de trafic est injuste et n'alloue pas suffisamment de bande passante pour les flux de données non critiques, favorisant trop largement les flux critiques. Assurer le respect des exigences temps réel tout en permettant la circulation des flux non critiques est le rôle des mécanismes d'ordonnancement de trafic.

Le CBS est un de ces mécanismes d'ordonnancement de trafic et est définit dans IEEE 802.1Qav [Q18]. Son but est d'empêcher des flux de données ayant une grande charge utile et un comportement propice aux transmissions en rafale de momentanément saturer des parties du réseau, ce qui pourrait conduire à une augmentation de la latence de bout en bout voire à des pertes de paquets pour les autres flux. Le CBS permet également d'empêcher le trafic ayant une priorité haute de bloquer le trafic ayant une priorité plus basse pendant une durée arbitraire.

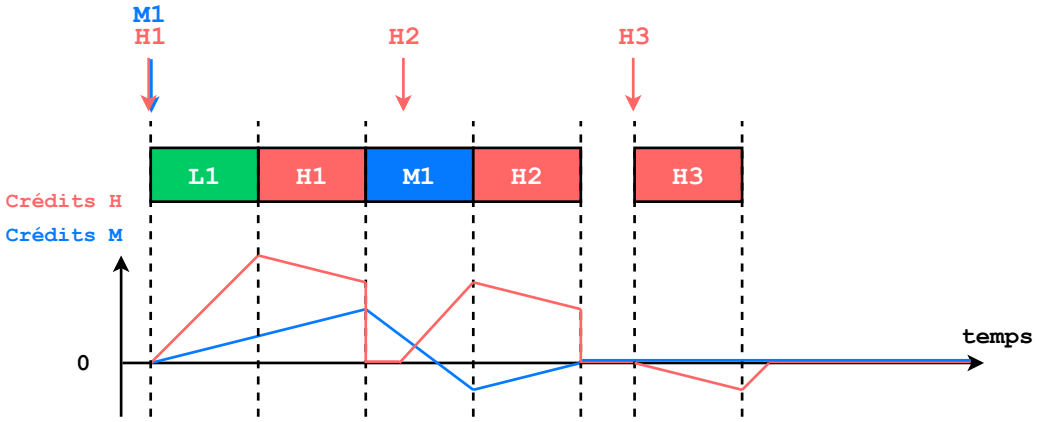


FIGURE 2.3 – Exemple du comportement du CBS.

Afin d'atteindre ce but, les différents flux de données sont répartis en plusieurs classes de trafic (trafic critique, audio, vidéo, « meilleur effort ») et, pour certaines de ces classes, un taux auxquelles elles gagnent des crédits peut leur être associé, appelé *idle slope*. Les trames appartenant à une classe de trafic ne peuvent être transmises que si la quantité de crédits associée à cette classe de trafic est positive ou nulle. Lorsqu'une trame est émise, la quantité de crédits associée à sa classe de trafic diminue à un taux égal à la différence entre la capacité en bande passante du port et de *idle slope*, appelé *send slope*. Quand des trames sont bloquées parce que la quantité de crédits associée à leur classe de trafic est strictement négative ou parce que des trames d'une autre classe de trafic sont en cours de transmission, la quantité de crédits associée à leur classe de trafic augmente au taux définit par *idle slope*. Ce comportement est illustré par la figure 2.3, qui présente deux classes de trafic H et M gérées par le CBS, H a un niveau de priorité supérieur à M qui a un niveau de priorité supérieur à L. Les flèches verticales indiquent le moment d'arrivée des trames et les rectangles leur transmission.

Lorsque la quantité de bande passante requise par une classe de trafic représente une proportion importante de la bande passante disponible, l'utilisation du CBS crée nécessairement des intervalles de temps pendant lesquels les trames de cette classe de trafic ne pourront pas être transmises ; elles devront attendre que la quantité de crédits associée à leur classe de trafic soit suffisante. Ce comportement est indésirable pour l'ordonnancement du trafic critique car il ne permet pas d'atteindre une gigue aussi faible que d'autres mécanismes d'ordonnancement, notamment le *Time Aware Shaper* (TAS).

2.2.3 *Time Aware Shaper*

Pour minimiser la latence de bout en bout et la gigue des flux de données critiques, les standards TSN, dans [Qbv16], rendent possible l'utilisation d'un mécanisme d'ordonnancement basé sur la division temporelle [7] : le TAS. Ce mécanisme d'ordonnancement de trafic définit un cycle, représenté par une durée fixe, et divise ce cycle en plusieurs fenêtres, ayant elles aussi une durée fixe. À chaque fenêtre est associée une liste contenant les classes de trafic dont les trames ont le droit d'être transmises pendant la durée de la fenêtre. Ce mécanisme permet d'isoler la transmission des flux de données critiques en créant des intervalles pendant lesquels la transmission leur est exclusivement réservée ; cela permet de garantir que les trames des flux de données critiques ne seront jamais bloquées par d'autres transmissions.

Cette division temporelle est illustrée dans la figure 2.2, par le tableau appelé GCL (*Gate Control List*). La GCL définit les fenêtres de transmission, leurs durées et la liste des classes de trafic dont les flux de données pourront être transmis pour chacune. Plusieurs classes de trafic peuvent partager une même fenêtre de transmission, dans ce cas, la sélection finale de la trame à transmettre se base sur la priorité de chaque trame. Cette priorité est définit par un PCP (*Priority Code Point*), dont la valeur est comprise entre 0 et 7 inclus, 0 étant la priorité la plus basse, c'est-à-dire la moins prioritaire, et 7 la plus élevée. Le PCP est inclus dans un des champs du préambule des trames Ethernet.

Définir une fenêtre temporelle pendant laquelle seules les trames appartenant à une classe de trafic spécifique peuvent être transmises n'est pas suffisant pour garantir que leur transmission aura bien lieu pendant cet intervalle de temps. En effet, il est possible qu'une trame ait commencé sa transmission avant le début de la fenêtre réservée et que son temps de transmission bloque la transmission des trames pour lesquelles la fenêtre a été réservée. Ce phénomène est illustré par la figure 2.4.

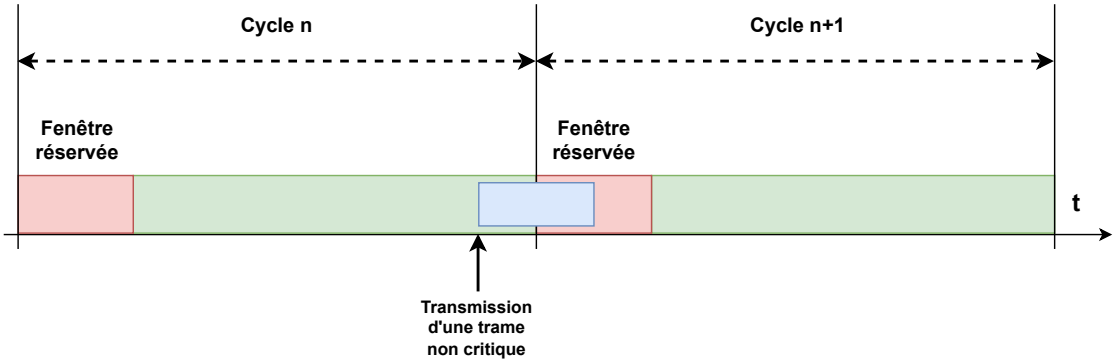


FIGURE 2.4 – Exemple de blocage d'une fenêtre protégée.

7. *Time-Division Multiple Access en anglais*

Afin d'empêcher ce phénomène de se produire, il est possible de définir des fenêtres temporelles ne permettant la transmission d'aucune trame, appelées *guard band*. Ces fenêtres doivent avoir une durée au moins égale au temps de transmission de la trame la plus volumineuse utilisant le port sur lequel la *guard band* est définie. La figure 2.5 illustre la protection offerte par une fenêtre de *guard band*, les durées ne sont pas à l'échelle.

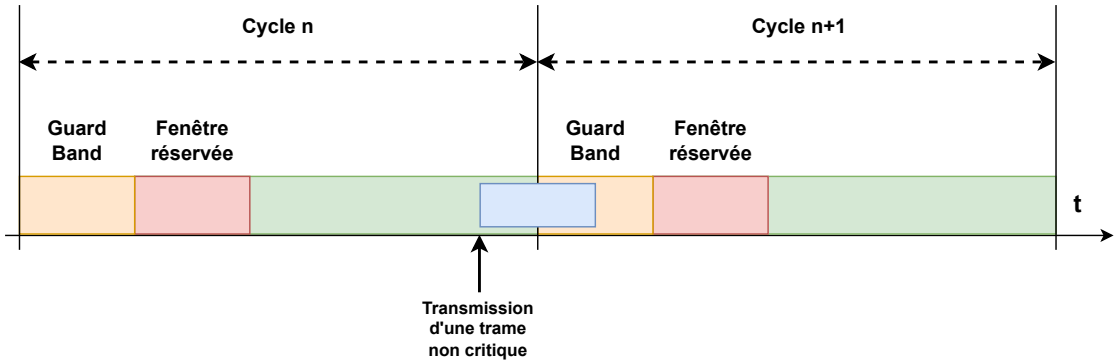


FIGURE 2.5 – Exemple de la protection apportée par une fenêtre de *guard band*.

Aucun nouvelle transmission ne peut débuter pendant une fenêtre de *guard band*. Afin de réduire le gâchis de bande passante engendré par ces fenêtres, le standard [Qbv16] définit le mécanisme appelé *length-aware scheduling* qui permet de déterminer la durée de transmission d'une trame. Si cette durée peut être contenue dans la fenêtre de *guard band* alors la trame peut être transmise. Néanmoins, ce mécanisme est incompatible avec l'utilisation d'un autre mécanisme : la transmission à la volée, ou *cut-through switching*. Cet autre mécanisme permet la transmission des trames avant même que leur en-tête ait été entièrement lu par un pont, c'est-à-dire dès que l'adresse de destination et l'interface de sortie ont été déterminées. La transmission à la volée est utile pour réduire la latence engendrée par le traitement des trames effectué par les ponts.

2.2.4 Préemption de trames

Le mécanisme de préemption de trames est spécifié par les standards [Qbu16] et [br16]. Ce mécanisme permet d'interrompre la transmission d'une trame en cours afin de débuter la transmission d'une trame d'un niveau de priorité supérieur.

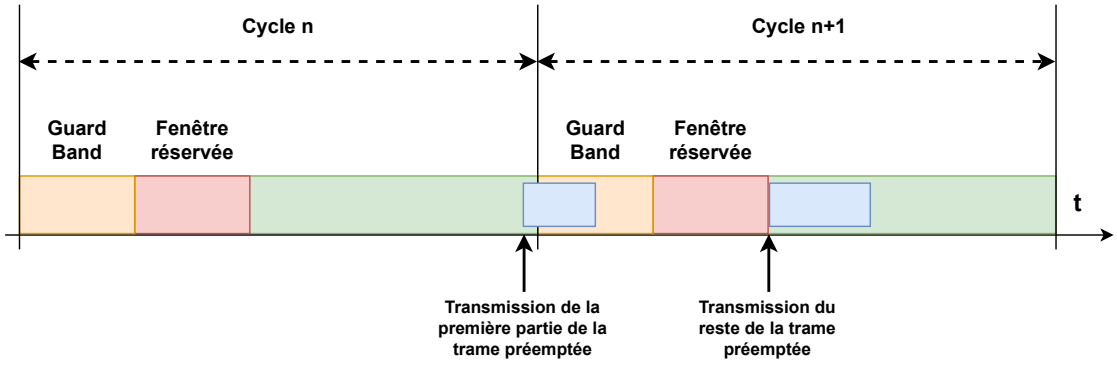


FIGURE 2.6 – Exemple d'utilisation du mécanisme de préemption de trame.

L'interruption d'une transmission est illustrée par la figure 2.6 où l'on peut voir une trame dont la transmission est interrompue puis reprise après la fin de la fenêtre protégée. Afin de pouvoir utiliser ce mécanisme, il est nécessaire que les deux nœuds entre lesquels la transmission a lieu le supportent. Il n'est pas possible qu'une trame qui en préempte une autre, c'est-à-dire qu'elle interromps la transmission de l'autre trame, soit elle-même interrompue, il ne peut y avoir qu'un seul niveau de préemption.

L'intérêt principal de ce mécanisme est l'optimisation de la taille des fenêtres de *guard band*. Ces fenêtres représentent un gâchis de bande passante car elle empêchent le commencement de nouvelles transmission pendant un intervalle de temps. L'utilisation du mécanisme de préemption de trame permet donc de réduire la durée des fenêtres de *guard band* à la plus faible valeur possible : le temps de transmission de la trame de la plus petite longueur qu'il n'est pas possible de préempter. La taille minimum d'une trame Ethernet étant de 64 octets, cette longueur est égale à cette taille minimum additionnée à la plus petite longueur qui n'est pas préemptable (car trop faible pour former une trame complète) : 63 octets. Grâce à l'utilisation de ce mécanisme, la durée des fenêtres de *guard band* peut donc être réduite au temps de transmission de $64 + 63 = 127$ octets.

2.2.5 Réplication et élimination de trames

Les standards TSN prévoient la possibilité de mettre en place des réseaux tirant parti de la redondance. Pour ce faire, le standard IEEE 802.1CB [CB17] définit un mécanisme de réplication et d'élimination de trames.

Ce mécanisme permet d'utiliser des chemins redondants pour la transmission de flux de données en émettant des copies d'une trame sur ses différents chemins puis en éliminant les trames supplémentaires de façon à ce qu'une seule d'entre elles arrivent à destination. La réplication de trame fonctionne par l'ajout d'un numéro de séquence aux trames qui permet de les différencier, un algorithme définit par le standard est ensuite en charge d'éliminer les trames devenues inutiles une fois le réseau traversé et la destination atteinte.

L'algorithme d'élimination de trames inclue également des capacités de sécurité et de détection de panne. L'élimination de trame n'est pas impactée par une réinitialisation du système entraînant une remise à zéro des numéros de séquences ou par un dysfonctionnement causant la transmission de plusieurs trames avec le même numéro de séquence. La détection de panne repose sur une hypothèse simple : pour n réplications d'une trame, il est attendu l'observation de $n - 1$ éliminations. Si cette hypothèse n'est pas respectée, une alerte peut être levée.

2.3 Conclusion

Dans ce chapitre, nous avons commencé par présenter les concepts fondamentaux d'Ethernet. Nous avons également abordé ses avantages et notamment la bande passante importante disponible dans un réseau Ethernet. Néanmoins, Ethernet commuté ne permet pas la mise en œuvre de réseaux temps réel car il ne permet pas de garantir le déterminisme du réseau et n'offre donc pas l'assurance du respect des exigences temps réel du système.

Nous avons présenté plusieurs des standards composants TSN. Ces standards concernent des aspects différents du réseau et jouent tous un rôle dans la conception d'un réseau temps réel. Ils permettent la synchronisation temporelle de l'ensemble des nœuds du réseau, l'ordonnancement du trafic et l'utilisation de la redondance au sein du réseau.

Il existe d'autres standards TSN que nous ne présentons pas ici car ils ne sont pas utilisés dans le cadre de nos travaux qui concernent la conception de système embarqués temps réel dont

les communications réseau sont centrées sur des flux de type contrôle-commande et multimédia. Différents profils ont également été définis par les groupes de travail TSN, ils regroupent des sous ensembles des standards TSN spécifiquement sélectionnés pour une utilisation dans des domaines particuliers, comme l'automobile ou l'avionique.

Le problème de configuration principal qui nous concerne est posé par les mécanismes d'ordonnancement de TSN. Dans le cadre de nos travaux, nous utilisons le TAS et le CBS conjointement ce qui rend d'autant plus complexe l'effort de configuration qu'il faut fournir pour respecter les exigences du système. L'utilisation d'une approche permettant de formaliser le problème afin d'en maitriser la complexité semble donc être nécessaire.

Chapitre 3

La modélisation

La modélisation permet la création d'un modèle du problème, c'est une étape qui peut permettre de grandement faciliter le processus amenant à la proposition d'une solution qui y répond. C'est par exemple le cas des plans d'une maison, qui sont une étape considérée comme indispensable dans le processus de construction. Une approche basée sur la modélisation est particulièrement utile dans le cas de problème complexe car elle permet de maitriser cette complexité en formalisant le problème.

Dans notre cas, la problématique est la conception et la configuration d'un réseau TSN. Une approche de modélisation pour ce problème doit donc permettre de créer une représentation des éléments du système.

Notre approche de modélisation allie la modélisation réseau, qui est une contribution de cette thèse qui sera présentée dans le chapitre 5, et la modélisation logicielle. La modélisation de l'architecture logicielle permet de représenter la partie du système qui produit les flux de données. Elle sera utilisée dans notre approche pour produire automatiquement un modèle des flux de données. Cette contribution sera présentée dans le chapitre 6.

La conception d'un réseau TSN nécessite l'utilisation d'outils tels que des simulateurs réseau. Ces outils ont besoin d'une définition du système simulé sous la forme d'un modèle de simulation. Afin de produire un tel modèle, il est nécessaire de pouvoir représenter à la fois la topologie du réseau et les flux de données qui sont échangés sur celui-ci. Notre approche permettant l'utilisation de plusieurs outils de simulation différents, il nécessaire de s'appuyer sur une modélisation du système contenant les informations requises pour créer les modèles utilisés par chacun de ces outils.

Dans un premier temps, nous donnerons une définition de ce qu'est un modèle. Nous présenterons ensuite l'approche de modélisation logicielle que nous utilisons dans le cadre des travaux présentés dans cette thèse, en commençant par donner un historique et une comparaison avec d'autres standards de modélisation logicielle. Enfin, nous présenterons les outils, principalement des simulateurs réseau, qui permettent la conception de réseau TSN et que nous utilisons dans nos travaux.

3.1 Qu'est-ce qu'un modèle

Les travaux présentés dans cette thèse s'appuient en très grande partie sur la notion de *modèle*. Le dictionnaire Le Robert donne la définition suivante :

> « Un modèle est une structure formalisée utilisée pour rendre compte d'un ensemble de phénomènes qui possèdent entre eux certaines relations. »

Comme cette définition l'indique, un modèle est une représentation simplifiée de la réalité permettant de faciliter les raisonnements. Dans le cadre de nos travaux, nous utilisons des modèles pour deux usages différents.

D'une part, nous montrons la construction de modèles pour étudier un phénomène réel – en l'occurrence, il s'agit de simuler et analyser le comportement du réseau tel qu'il existe. Cet aspect est abordé dans le chapitre 5.

D'autre part, un modèle peut être synonyme de plan : il s'agit de saisir les informations importantes du système en vue de le construire effectivement. Nous manipulons des modèles selon cette acception dans le chapitre 6. On parle alors d'ingénierie dirigée par les modèles (*model driven engineering*).

Dans les deux cas, les modèles que nous manipulons sont représentés selon un formalisme lié au standard UCM, que nous détaillons dans la section 3.2.

La notion de métamodèle sera utilisée dans la suite de ce manuscrit. D'après [SV06] : « Un métamodèle est un modèle qui décrit la structure de modèle ». Une approche de modélisation logicielle se basant sur des composants pour représenter les applications doit donc s'appuyer sur un métamodèle définissant ce qu'est un composant.

3.2 Modélisation basée sur UCM

UCM est un standard de l'OMG [8] auquel Thales a contribué. Le nom complet d'UCM est *Unified Component Model for Distributed Real-Time Embedded Systems*. UCM vise à proposer une solution unifiée pour décrire des composants permettant de concevoir des applications logicielles à destination de systèmes répartis temps-réel embarqués.

Au cours des années, l'OMG a définit les standards *Common Object Request Broker Architecture* (CORBA) [COR21] et *Interface Definition Language* (IDL) [IDL18] pour la construction de systèmes d'information répartis.

Le standard CCM [CCM06] est un modèle de composants pour les systèmes d'information, il s'appuie sur CORBA pour les communications. Toutefois, il a été constaté dans les pratiques industrielles (notamment au sein de Thales) qu'il n'y a pas de solution de communication universelle. Par conséquent, une évolution de ses standards, n'étant pas limitée à CORBA, est nécessaire.

CCM s'attache à décrire les composants essentiellement par leurs interfaces ; il ne décrit pas la sémantique d'exécution des composants, qui est laissée à la charge des développeurs. Cela empêche son utilisation pour la conception de systèmes embarqués temps réel qui nécessite une spécification explicite de ces informations.

UCM se place dans la continuité de CCM et s'en distingue notamment par la possibilité de spécifier la sémantique d'exécution des composants et des communications (par opposition à l'adhérence de CCM à CORBA).

UCM peut s'exprimer dans le langage UML, Un prototype a été développé pour le modeleur UML Papyrus [Pap17] en se basant sur UCM version 1.0.

Le standard UCM en lui-même se limite à la définition des composants ; il ne traite pas de leur déploiement, pour lequel l'OMG propose déjà le standard Deployment & Configuration [Dan06]. Il ne propose pas non plus de description du comportement interne des composants, afin d'éviter de restreindre les possibilités (comme CCM s'est restreint par une adhérence à CORBA). UCM est conçu pour s'exprimer facilement en UML [UML17], permettant ainsi de décrire le comportement des composants à l'aide de diagrammes UML.

8. https ://www.omg.org/spec/UCM

Thales a développé une implémentation d'UCM sous la forme d'un prototype destiné aux expérimentations. Cet outil – auquel nous avons contribué – s'appelle Sigil-UCM [Ver23]. Il implémente le standard UCM lui-même et le complète en y ajoutant un mécanisme de déploiement et la possibilité de spécifier le comportement des composants.

Dans la suite de cette section, nous présentons les concepts d'UCM ainsi que les éléments de déploiement implémentés dans Sigil-UCM. La spécification du comportement des composants est abordée dans le chapitre 6 car elle a été développée en lien direct avec notre travail de thèse. Les différents exemples sont des extraits de l'architecture logicielle que nous avons développée pour évaluer notre travail, et qui est expliquée complètement au chapitre 9.

3.2.1 Le modèle de composants d'UCM

Le modèle de composant lui-même est spécifié par le chapitre 9 du standard [UCM21]. Nous en décrivons ici les éléments essentiels.

UCM est un modèle de composants qui permet une séparation stricte entre les algorithmes d'une application et la plateforme technique supportant leur exécution. Il s'agit d'éviter que le code logiciel des algorithmes ait une dépendance explicite vis-à-vis des caractéristiques de la plateforme d'exécution. Typiquement, un code de contrôle de rotation d'un moteur ne devrait pas dépendre de la bibliothèque de communication réseau utilisée pour la réception de la consigne.

Une telle isolation entre partie fonctionnelle (le code de contrôle du moteur) et la partie non-fonctionnelle (la gestion des communications réseau) facilite l'utilisation du code dans différentes configuration (utilisation d'une autre bibliothèque de communication, intégration dans un banc de test, etc.).

Cette approche conduit à encapsuler le code algorithmique (également appelé « code métier ») dans une armature logicielle qui, elle, dépend des bibliothèques du système d'exploitation, voire interagit directement avec le matériel.

Une application UCM est ainsi constituée d'une armature technique qui contrôle l'exécution des algorithmes. UCM permet de spécifier le paramétrage de l'armature ainsi que les interfaces avec le code algorithmique. Il est possible de spécifier une application en UCM de façon suffisamment précise pour que la modélisation de l'application contienne toutes les informations nécessaires pour produire le code logiciel de l'armature technique qui pilotera le code algorithmique.

Dans cette section, nous baserons notre explication sur un système exemple représentant une architecture très simplifiée de logiciel de vol d'un drone. Cet exemple servira d'illustration dans les différents chapitres. Le logiciel de vol permet de contrôler la rotation des moteurs en fonction de la trajectoire à suivre.

Le système comporte plusieurs composants, dont les plus importants sont les composants *controle* et *moteur* qui sont en charge respectivement de la navigation du drone par envoi de consignes aux moteurs et du respect de ces consignes ainsi que de la transmission de l'état courant du moteur. Ces deux composants sont déployés sur des nœuds différents et communiquent en permanence par le biais d'un réseau TSN.

Un composant encapsule les parties métier de l'application, c'est-à-dire les algorithmes. En UCM, la déclaration d'un composant permet de spécifier les interactions de ce composant avec d'autres composants, ainsi que les interactions de ce composant avec son environnement d'exécution. Les interactions entre composants spécifient la façon dont les données sont échangées (le type des données, le protocole d'échange, etc.). Les interactions avec l'environnement d'exécution spécifient les moyens sur lesquels chaque composant s'appuie pour réaliser son exécution (par exemple pour l'appel à des services techniques tels qu'une horloge, ou la spécification d'un

déclenchement périodique).

3.2.2 Modélisation des applications sous forme de composants

Déclaration d'un composant UCM de gestion de moteur

La définition d'un composant se fait en deux parties : la déclaration de son type et d'une ou plusieurs implémentations. Le type de composant décrit les ports d'interactions avec les autres composants ; les implémentations spécifient les interactions avec l'environnement d'exécution.

```
1 <compType name="moteur">
2   <port bindings="consigne_b" name="consigne" type="::ucm_core::messages
    ↪ ::msg_rcvr_pt"/>
3   <port bindings="etat_b" name="etat" type="::ucm_core::messages::
    ↪ msg_emtr_pt"/>
4 </compType>
```

Listing 3.1 – Déclaration du type de composant *Moteur*.

Le listing 3.1 présente la déclaration du type de composant *Moteur*. Ce type de composant comporte deux ports d'interaction : le port *consigne*, dont le rôle est de recevoir la nouvelle consigne de vitesse qu'il faut respecter, et le port *etat*, dont le rôle est de transmettre l'état actuel du moteur au composant qui le contrôle.

Chaque déclaration de port est caractérisée par un nom, un type de port et les types des données manipulées. Un type de port spécifie les interfaces que le composant doit fournir ou qu'il requiert pour communiquer (cf. listings 3.2 et 3.3).

```
1  <platformModule name="ucm_core">
2   <interactionModule name="messages">
3    [...]
4    <portType name="msg_emtr_pt" role="::ucm_core::roles::producer">
5     <portElement interface="api::msg_emtr_intf" kind="required" name="
     ↪ emtr_pe"/>
6    </portType>
7    <portType name="msg_rcvr_pt" role="::ucm_core::roles::consumer">
8     <portElement interface="api::msg_rcvr_intf" kind="provided" name="
     ↪ rcvr_pe"/>
9    </portType>
10   [...]
11  </interactionModule>
12 </platformModule>
```

Listing 3.2 – Définition de types de ports.

Les interfaces font référence à des méthodes, les paramètres de ces méthodes peuvent porter des types de données complètement définis, ou bien des types *abstraits*. Le listing 3.4 donne des exemples de déclarations de types de données : le type `transm_t` est un type abstrait tandis que le type énuméré `return_codes` est concret.

Les composants ne peuvent pas manipuler de types de données abstraits. Lorsqu'une méthode d'interface ne manipule que des types de données concrets, elle peut être manipulée telle quelle

```
<platformModule name="ucm_core">
 <interactionModule name="messages">
  <contractModule name="api">
   [...]
   <interface name="msg_emtr_intf">
    <method name="push">
     <param dir="in" name="message" type="transm_data_t"/>
     <param dir="return" name="ecode" type="::ucm_core::return_codes::
    ↪ comm_ecode"/>
    </method>
   </interface>
   <interface name="msg_rcvr_intf">
    <method name="push">
     <param dir="in" name="message" type="transm_data_t"/>
    </method>
   </interface>
   [...]
  </contractModule>
 </interactionModule>
</platformModule>
```

Listing 3.3 – Définition d'interfaces.

```
<platformModule name="ucm_core">
 <interactionModule name="messages">
  <contractModule name="api">
   <abstractDataType name="transm_data_t"/>
   [...]
  </contractModule>
 </interactionModule>
 [...]
 <contractModule name="return_codes">
  <enum indexType="int16" name="comm_ecode">
   <value index="0" name="ok"/>
   <value index="1" name="internal_error"/>
   <value index="2" name="comm_error"/>
  </enum>
 </contractModule>
</platformModule>
```

Listing 3.4 – Définition de types de données.

par le composant. En revanche, lorsqu'une méthode d'interface manipule des types de donnés abstraits, il est nécessaire d'associer ces types abstraits à des types concrets au moment de déclarer le port du composant. Le listing 3.5 illustre une telle association.

Les ports d'un composant sont les points d'interaction de ce composant avec les autres composants. Une modélisation UCM décrit complètement les méthodes qui encapsulent le code algorithmique. Cela permet de savoir, pour chaque composant, quelle méthode il devra implémenter et quelles méthodes il pourra appeler.

Différentes implémentations de composants peuvent être associées à un type de composant

```
<bindingSet name="etat_b">
 <binding abstract="::ucm_core::messages::api::transm_data_t" actual="
   ↪ types::etat_t"/>
</bindingSet>
<bindingSet name="consigne_b">
 <binding abstract="::ucm_core::messages::api::transm_data_t" actual="
   ↪ types::consigne_t"/>
</bindingSet>
```

Listing 3.5 – Déclaration d'association de type

```
<atomic lang="::ucm_lang::cpp::CPP11_typed" name="moteur_atomic" type="
   ↪ ::model::comps::moteur">
  <policyRef ref="exec_psv"/>
</atomic>
```

Listing 3.6 – Implémentation atomique du composant *moteur_atomic* du type de composant *Moteur*.

donné. Une implémentation peut être *composite* si elle consiste en une décomposition en sous-composants, ou *atomique* si elle encapsule directement du code algorithmique.

Le listing 3.6 présente une implémentation atomique du type de composant *Moteur* du listing 3.1. Cette implémentation atomique spécifie que le composant sera programmé dans le langage C++ 11. Elle indique aussi que le composant est associé à une politique technique *exec_psv*, déclarée dans le listing 3.7. La politique technique *exec_psv* spécifie que le composant est susceptible d'être exécuté plusieurs fois simultanément. Les politiques techniques seront présentées plus en détails dans la sous-section 3.2.3.

```
<policy def="::pharos_lib::exec::prs_pasv" name="exec_psv">
 <componentRef ref="controle1"/>
 <componentRef ref="moteur1"/>
 [...]
</policy>
```

Listing 3.7 – Politique d'exécution passive non protégée.

À partir de la déclaration d'une implémentation atomique de composant, il est possible de déduire un squelette de code pouvant accueillir le code algorithmique (listing 3.8). L'implémentation atomique spécifie en effet toutes les informations nécessaires, à savoir les différentes signatures de méthodes et le langage de programmation visé (ici C++ 11).

Le listing 3.8 présente l'implémentation en C++ 11 de la méthode *push* du composant *moteur_atomic*. Le corps de cette méthode, après avoir réalisé des calculs dont le code a été remplacé par un simple commentaire pour des raisons de clarté, permet de recevoir la nouvelle consigne transmise au moteur par le composant de contrôle.

Nous pouvons remarquer que le code de la méthode *push* ne fait aucune hypothèse sur la façon dont elle reçoit le message (de type *consigne_t*, ni sur l'entité à qui elle transmet le résultat de son calcul (de type *etat_t*).

```
1 void moteur_atomic_consigne_rcvr_pe::push (::consigne_b::
    ↪ ucm_core_messages::consigne_t const& message) {
2  // Start of user code consigne_rcvr_pe push
3  ::etat_b::ucm_core_messages::etat_t etat_moteur;
4
5  // calcul de etat_moteur en fonction de message (de type consigne_t)...
6
7  this->context->get_etat_emtr_pe()->push(etat_moteur);
8  // End of user code
9 }
```

Listing 3.8 – Implémentation en C++ de la méthode *push* du composant *moteur_atomic*.

Déclaration d'un composant UCM de contrôle

Nous décrivons ici un autre composant dont le rôle est de calculer la consigne à envoyer à un moteur en fonction de l'état courant de ce moteur et d'un ordre de mission. Ce composant possède plus de ports, mais le principe est le même.

```
1 <compType name="controle">
2   <port bindings="consigne_b" name="consigne_moteur" type="::ucm_core::
    ↪ messages::msg_emtr_pt"/>
3   <port bindings="etat_b" name="etat_moteur" type="::ucm_ext_interac::
    ↪ shared_data::sd_reader_pt"/>
4   <port bindings="coord_b" name="consigne_mission" type="::
    ↪ ucm_ext_interac::shared_data::sd_reader_pt"/>
5 </compType>
```

Listing 3.9 – Déclaration du type de composant *Controle*.

Le listing 3.9 présente la déclaration du type de composant *Controle*. Ce type de composant comporte trois ports d'interaction : *consigne_moteur*, dont le rôle est de transmettre la nouvelle consigne de vitesse à respecter au moteur, *etat_moteur*, dont le rôle est de recevoir l'information sur l'état actuel du moteur, et *consigne_mission*, dont le rôle est de recevoir les nouvelles coordonnées à atteindre qui serviront à calculer la nouvelle consigne de vitesse à transmettre au moteur.

De la même façon que pour le composant *Moteur*, les ports sont associés à des types de port et des associations de types de données. Ce composant utilise des types de ports différents (cf. listing 3.10), conçu pour un mécanisme de lecture de données plutôt que pour la consommation de messages.

De la même façon que pour le composant *Moteur*, le composant *Controle* est spécifié par une implémentation atomique, représentée sur le listing 3.11. Cette implémentation fait référence à la même politique technique *exec_psv* que pour *moteur1*, ainsi qu'à une politique d'exécution périodique *exec_periodique*, déclarée au listing 3.12. Cette politique précise la période, la phase et la priorité de l'exécution périodique. Dans sa définition, la politique technique *exec_periodique* spécifie une méthode `run` qui doit être implémentée par le composant.

Le listing 3.13 présente l'implémentation en C++ 11 de la méthode *run* du composant *controle_atomic*. Le corps de cette méthode réalise quelques calculs puis transmet la nouvelle consigne de vitesse à respecter au composant du moteur.

```
<platformModule name="ucm_ext_interac">
 <interactionModule name="shared_data">
  <contractModule name="api">
   <interface name="data_reader">
    [...]
    <method name="read_data">
     <param dir="out" name="data" type="::ucm_core::messages::api::
    ↪ transm_data_t"/>
     <param dir="return" name="ecode" type="::ucm_core::return_codes::
    ↪ comm_ecode"/>
    </method>
   </interface>
  </contractModule>
  [...]
  <portType name="sd_reader_pt" role="::ucm_core::roles::consumer">
   <portElement interface="api::data_reader" kind="required" name="
   ↪ rdr_pe"/>
   <portElement interface="api::data_notification" kind="provided" name=
   ↪ "notif_pe"/>
  </portType>
  [...]
 </interactionModule>
</platformModule>
```

Listing 3.10 – Définition de types de ports.

```
<atomic lang="::ucm_lang::cpp::CPP11_typed" name="controle_atomic" type=
   ↪ "::model::comps::controle">
  <policyRef ref="exec_psv"/>
  <policyRef ref="exec_periodique"/>
</atomic>
```

Listing 3.11 – Implémentation atomique du composant *controle_atomic* du type de composant *Controle*.

```
<policy def="::pharos_lib::trig::prs_periodic" name="exec_periodique">
 <componentRef ref="controle1"/>
 [...]
 <config def="psec_offset" value="{0,ms}"/>
 <config def="psec_period" value="{30,ms}"/>
 <config def="psec_priority" value="4"/>
</policy>
```

Listing 3.12 – Politique d'exécution périodique.

3.2.3 Bibliothèques de plateformes

Dans le listing 3.6, l'implémentation atomique présentée fait référence à la politique technique *exec_psv* qui est une politique technique d'exécution passive. Cette politique technique indique

```
1  void controle1_exec_periodique_activation::run () {
2   // Start of user code exec_periodique_activation run
3   ::etat_b::ucm_core_messages::etat_t etat_moteur;
4   ::coord_b::ucm_core_messages::coord_t mission;
5   ::consigne_b::ucm_core_messages::consigne_t consigne_moteur
6
7   // acces aux ports du composant
8   ::etat_b::ucm_ext_interac_shared_data_api::data_reader*
      ↪ lecture_etat_moteur = get_etat_moteur_rdr_pe ();
9   ::consigne_b::ucm_core_messages_api::msg_emtr_intf*
      ↪ ecriture_consigne_moteur = get_consigne_moteur_emtr_pe ();
10  ::coord_b::ucm_ext_interac_shared_data_api::data_reader*
      ↪ lecture_mission = get_consigne_mission_rdr_pe ();
11
12  lecture_etat_moteur->read_data(etat_moteur);
13  lecture_mission->read_data(mission);
14
15  // calcul de consigne_moteur...
16
17  ecriture_consigne_moteur->push(consigne_moteur);
18  // End of user code
19 }
```

Listing 3.13 – Implémentation en C++ de la méthode *run* du composant *controle_atomic*.

que les composants auxquels elle s'applique ne déclenche pas leur exécution d'eux-mêmes mais le font suite à un stimulus extérieur.

Dans le listing 3.11, c'est une politique technique appelée *exec_periodique* qui est utilisée. Cette politique technique implique que les composants auxquels elle s'applique s'exécutent de façon périodique et qu'il est possible de spécifier leur période et leur phase d'exécution.

D'autre part, dans les listings 3.1 et 3.9 les composants déclarent des ports qu'il est nécessaire de connecter les uns aux autres pour transporter les données d'un composant à un autre. L'entité qui connecte ces ports et qui assure ce transport est appelé un *connecteur*.

Les politiques techniques déterminent la sémantique de services offerts aux composants pour interagir avec l'environnement d'exécution, un déclenchement périodique ou une exécution passive (potentiellement simultanée), par exemple.

Les connecteurs déterminent la sémantique des échanges entre les ports des composants, par exemple l'utilisation d'un protocole réseau donné, d'un format de sérialisation des données, etc.

Selon le standard UCM, les définitions des connecteurs et des politiques techniques, de même que les langages d'implémentation disponibles, forment une *plateforme* – par contraste avec les composants qui forment l'*application*.

Les éléments de la plateforme sont définit dans des bibliothèques de modèles. Dans un processus de modélisation en UCM, l'architecte logiciel déclare les types de données de son application et spécifie les composants en s'appuyant sur les définitions de politiques techniques et de types de ports d'une plateforme donnée. Les composants peuvent ensuite être connectés les uns aux autres en utilisant des connecteurs faisant référence à des définitions de cette même plateforme.

Dans son chapitre 13, le standard UCM [UCM21] définit une bibliothèque standard, qui comprend des déclarations de politiques techniques, de connecteurs et de types de ports, ainsi qu'une description des sémantiques d'exécution et de communication associées. Les listings 3.3,

3.4, 3.2 et 3.10 sont extraits de cette bibliothèque standard. La bibliothèque standard déclare aussi le langage C++ 11 pour l'implémentation des composant ; les règles de traduction associées sont décrites dans le chapitre 16.

La bibliothèque standard définit des types de port et des connecteurs pour des communications par message, par donnée partagée, par consommation de donnée, par requête-réponse et par appel de service. Elle définit également des politiques techniques (avec les interfaces associées) pour des services d'horloge, de *log*, de déclenchement périodique, d'exécution passive non-protégée, passive protégée et active protégée.

Elle fournit des définitions générales, comme la politique technique d'exécution représentée au listing 3.14. Cette politique technique spécifie que le composant doit implémenter une méthode `run` qui sera appelée par la politique technique, mais ne précise pas les conditions de cet appel.

```
<platformModule name="ucm_core">
 <policyModule name="comp_exec">
  <contractModule name="api">
   <interface name="comp_exec_intf">
    <method name="run"/>
   </interface>
  </contractModule>
  <policyDef applicability="on_component_only" aspect="comp_trig_asp"
   ↪ name="self_exec_comp">
   <comment>self-executing component</comment>
   <portElement interface="api::comp_exec_intf" kind="provided" name="
   ↪ activation"/>
  </policyDef>
  [...]
 </policyModule>
</platformModule>
```

Listing 3.14 – Définition de la politique technique standard d'exécution périodique.

Différentes politiques techniques étendent celle-ci. C'est le cas par exemple de la politique de déclenchement périodique (listing 3.15).

```
<policyModule name="ucm_ext_exec">
 <policyDef applicability="on_component_only" aspect="::ucm_core::
   ↪ comp_exec::comp_trig_asp" name="prdc_self_exec_comp">
  <comment>periodic self-executing component. It will invoke method run
   ↪ every period.</comment>
  <extends ref="::ucm_core::comp_exec::self_exec_comp"/>
  <configParam name="psec_period" type="contracts::ucm_duration_t"/>
  <configParam name="psec_priority" type="contracts::priority_t"/>
  <configParam name="psec_offset" type="contracts::ucm_duration_t"/>
 </policyDef>
</policyModule>
```

Listing 3.15 – Définition de la politique technique standard d'exécution périodique.

Les connecteurs de la bibliothèque standard sont définis par une sémantique générale. Par exemple, le connecteur de communication par message est représenté au listing 3.16. Il spécifie

les interfaces pour les composants mais ne précise pas quelle bibliothèque de communication utiliser pour transmettre les données.

```
<platformModule name="ucm_core">
 <interactionModule name="messages">
  [...]
  <connectorDef name="simple_msg_cnt" pattern="msg_intr_pat">
   <connPort name="emitter" role="emitter" type="msg_emtr_pt"/>
   <connPort name="receiver" role="receiver" type="msg_rcvr_pt"/>
   [...]
  </connectorDef>
  [...]
 </interactionModule>
</platformModule>
```

Listing 3.16 – Définition du connecteur standard de communication par message.

Le rôle de la bibliothèque standard d'UCM est de définir des interfaces et des sémantiques de base pour assurer la portabilité des composants. En effet, un composant UCM dont la déclaration se base uniquement sur les définitions de la bibliothèque standard sans faire d'hypothèse supplémentaire est censé pouvoir être déployé sur n'importe quelle plateforme logicielle qui se conformerait à la bibliothèque standard.

Néanmoins, l'usage normal d'UCM est d'étendre la bibliothèque standard pour spécialiser la sémantique en fonction de la technologie de la plateforme. Dans le cadre de nos travaux, nous utilisons une plateforme appelée *Pharos*, développée à Thales pour les expérimentations TSN. Cette plateforme est décrite dans l'annexe B.

3.2.4 Modélisation du déploiement dans Sigil-UCM

Afin d'obtenir une architecture logicielle complète, il est nécessaire de spécifier comment les composants sont *déployés*, c'est-à-dire comment ils sont connectés entre eux et comment ils sont regroupés dans des binaires, qui eux-mêmes s'exécuteront sur des machines physiques.

Dans le cadre de nos travaux, nous déployons les composants en utilisant les fonctionnalités de l'outil Sigil-UCM développé à Thales. Cet outil complète UCM en fournissant notamment le moyen de spécifier l'assemblage et l'allocation des composants de l'application sur des nœuds d'exécution, ainsi que le paramétrage de la plateforme.

La spécification du déploiement des composants d'une application UCM avec Sigil-UCM se fait en deux étapes. Tout d'abord la spécification d'un *plan d'application* qui décrit la façon dont les composants sont assemblés, puis la spécification d'un ou plusieurs *plans d'allocation* qui décrivent l'allocation des composants sur des nœuds d'exécution et le paramétrage des politiques techniques et des connecteurs.

Le listing 3.17 illustre un plan d'application qui comprend un exemplaire du composant *Moteur* et un exemplaire du composant *Controle*. Un plan d'application contient deux partie : l'assemblage et l'ensemble des exemplaires de composants (*instances*). L'assemblage spécifie que les deux composants sont reliés l'un à l'autre par le connecteur de message texttt-simple_msg_cnt et le connecteur de données partagées `sd_cnt` de la bibliothèque UCM standard. Le port *consigne_mission* du composant `controle1` est relié à un connecteur bouchon défini dans la bibliothèque Pharos.

```
<appliPlan name="exemple">
 <appAssembly name="ex1">
  <part name="ctrl" ref="::model::impl::controle_atomic"/>
  <part name="mtr" ref="::model::impl::moteur_atomic"/>
  <connection name="consigne_conn" ref="::ucm_core::messages::
   ↪ simple_msg_cnt">
   <end name="controle_consigne_moteur" part="ctrl" port="
   ↪ consigne_moteur"/>
   <end name="moteur_consigne" part="mtr" port="consigne"/>
  </connection>
  <connection name="etat_conn" ref="::ucm_ext_interac::shared_data::
   ↪ sd_cnt">
   <end name="moteur_etat" part="mtr" port="etat"/>
   <end name="controle_etat_moteur" part="ctrl" port="etat_moteur"/>
  </connection>
  <connection name="ordre_mission" ref="::pharos_dista::comm::
   ↪ prs_stub_connector">
   <end name="controle_consigne_mission" part="ctrl" port="
   ↪ consigne_mission"/>
  </connection>
 </appAssembly>
 <instanceSet>
  <compInst name="ctrl" ref="ex1.controle1"/>
  <compInst name="mtr" ref="ex1.moteur1"/>
  <connInst name="consigne_conn" ref="ex1.consigne_conn">
   <endInst compInst="ctrl" name="controle_consigne_moteur" port="
   ↪ consigne_moteur"/>
   <endInst compInst="mtr" name="moteur_consigne" port="consigne"/>
  </connInst>
  <connInst name="etat_conn" ref="ex1.etat_conn">
   <endInst compInst="mtr" name="moteur_etat" port="etat"/>
   <endInst compInst="ctrl" name="controle_etat_moteur" port="
   ↪ etat_moteur"/>
  </connInst>
  <connInst name="ordre_mission" ref="ex1.ordre_mission">
   <endInst compInst="ctrl" name="controle_consigne_mission" port="
   ↪ consigne_mission"/>
  </connInst>
 </instanceSet>
</appliPlan>
```

Listing 3.17 – Exemple de plan d'application

La figure 3.1 illustre ce plan d'application qui lie les ports de deux composants entre eux à l'aide de deux connecteurs.

L'ensemble des instances est la mise à plat de l'assemblage. Étant donné que les composants (`moteur_atomic` et `controle_atomic`) sont des implémentations atomiques, l'ensemble des instances décrit la même chose que l'assemblage. Si certains composants avaient été composites, l'ensemble des instances aurait représenté la mise à plat de l'architecture.

Un plan d'allocation doit compléter un plan d'application pour spécifier sur quels nœuds sont déployés les instances de composants. Le listing 3.18 décrit un plan d'allocation simple,

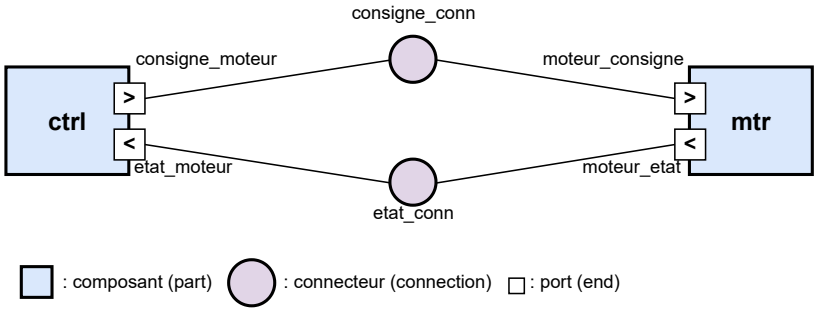


FIGURE 3.1 – Schéma d'un exemple de plan d'application.

dans lequel les deux instances de compoants `mtr` et `ctrl` déclarés dans le plan d'application sont alloués à un même nœud `local_test_bench`. Le plan d'allocation précise les définitions – et donc les sémantiques – des éléments de plateforme utilisés. Ainsi, la politique technique est raffinée pour préciser que nous utilisons la politique de déclenchement périodique fournie par la plateforme Pharos. De la même façon, les deux connecteurs sont raffinés pour utiliser des connecteurs de communication locale (c'est-à-dire sans utilisation du réseau) de la plateforme Pharos. La déclaration du nœud `local_test_bench` fait référence à la définition du nœud banc de test fourni par la plateforme Pharos.

```
1  <allocPlan name="ex1_alloc">
2   <appliPlanRef ref="::exemple"/>
3   <compNode def="::pharos_dista::rsc::prs_dst_process"lang="::ucm_lang::
      ↪ cpp::CPP11_typed" name="local_test_bench">
4    <compInstRef ref="ctrl"/>
5    <compInstRef ref="mtr"/>
6    <config def="udp_port" value="1234"/>
7   </compNode>
8   <policyConf name="exec_periodique_controle" policy="exec_periodique">
9    <compInstRef ref="ctrl"/>
10   <refinedIn ref="::pharos_lib::trig::prs_periodic"/>
11  </policyConf>
12 </allocPlan>
```

Listing 3.18 – Exemple de plan d'allocation

Nos expérimentations sont décrites dans le chapitre 9, qui contient des listings représentant des plans d'application et d'allocation utilisant la plateforme Pharos.

Dans le cas où il serait nécessaire de décrire l'environnement matériel des nœuds d'exécution, Sigil-UCM permet de décrire des *environnements* regroupant des *ressources* de communication et d'exécution. Nous utilisons ces notions pour décrire les éléments de modélisation réseau dans le chapitre 5.

3.2.5 Modèle d'implémentation standard d'UCM

UCM définit trois catégories d'entités principales : les composants, les politiques techniques et les connecteurs. Le standard indique comment déclarer des composants (cf. section 3.2.2) en s'appuyant sur des définitions de politiques techniques et de connecteurs formant une plateforme

(cf. section 3.2.3). Dans la section 3.2.4, nous avons brièvement présenté la spécification du déploiement des composant avec l'outil Sigil-UCM. La combinaison de tous ces éléments permet de produire une armature logicielle [9] pour exécuter le code algorithmique des composants.

Le standard UCM définit la façon dont est structurée une application, c'est-à-dire qu'il définit l'organisation de l'armature logicielle responsable de la gestion des traitements métier.

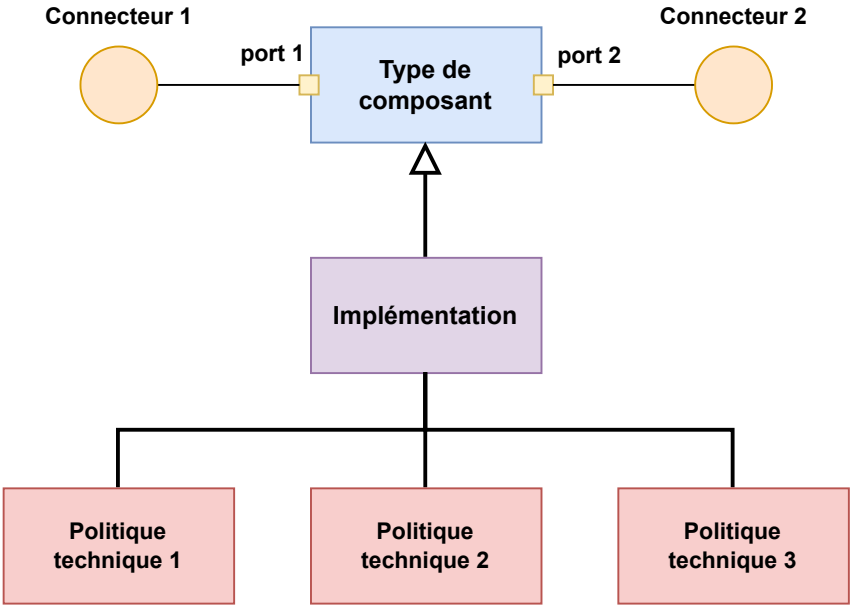


FIGURE 3.2 – Réunion d'un type de composant, d'une implémentation atomique, des politiques techniques et des connecteurs associés aux ports

Cette armature logicielle est construite à partir de l'assemblage des politiques techniques, des connecteurs et des ports qui les relient aux autres composants, comme repréenté sur la figure 3.2. Chacun de ces éléments (implémentation atomique, politique technique, fragment de connecteur associé à un port) sont traduits par des *microcomposants* reliés les uns aux autres par l'intermédiaire des interfaces requises et fournies. Le cycle de vie des microcomposants (création, connexion, déconnexion, destruction) est géré par un *conteneur*.

3.2.6 Conclusion

Dans cette section, nous avons présentés les éléments du standard UCM sur lesquels notre travail s'appuie. L'un des principes fondamentaux d'UCM est de permettre une séparation stricte entre les éléments algorithmiques d'une application et les éléments techniques liés à la plateforme d'exécution.

Nous avons expliqué comment décrire les éléments d'une architecture logicielle en utilisant les trois entités principales d'UCM (composants, politiques techniques, connecteurs) et nous avons présenté comment l'outil Sigil-UCM développé au sein de Thales complète le standard UCM pour décrire le déploiement des composants et ainsi permettre la création d'une armature logicielle encapsulant le code algorithme des compoants. Cette approche permet de redéployer les composants selon différentes configurations (décrites par les plans d'application et d'alloca-

9. *framework* en anglais

tion) et ainsi produire des armatures logicielles permettant l'exécution les codes applicatifs des composants en fonction des paramètres spécifiés dans les plans d'allocations.

Nos travaux s'appuient sur UCM, combiné aux compléments de Sigil-UCM, pour les différentes modélisations dans les chapitres suivants.

3.3 Outils de conception et de configuration

Il existe plusieurs outils pour assister la conception et la configuration de réseaux TSN, notamment en permettant de les simuler avant de déployer leur configuration sur de l'équipement réel.

NeSTiNg [FHC+19] est une infrastructure logicielle de simulation de réseaux TSN qui supportent plusieurs des standards les plus répandus, notamment IEEE 802.1Qav, IEEE 802.1Qbv et IEEE 802.1Qbu. CoRE4INET est une autre infrastructure logicielle de simulation qui a originellement été développé pour simuler des réseaux TTEthernet [SBH+09]. CoRE4INET peut désormais simuler des réseaux TSN et supporte les standards suivants : IEEE 802.1Qav, IEEE 802.1Qbv, IEEE 802.1Qci. TSimNet [HGO16] est une troisième possibilité pour la simulation de réseaux TSN mais ne supporte pas de mécanismes d'ordonnancement de trafic comme le *Time Aware Shaper* (TAS) ou le *Credit-Based Shaper* (CBS).

Ces trois infrastructures de simulation de réseaux TSN reposent sur le simulateur OMNeT++ [10] qui est un simulateur à évènements discrets, et sur son modèle de simulation de réseaux, INET [11]. L'utilisation combinée d'OMNeT++ et d'INET pour la simulation réseau est largement répandue.

Il est important de noter que NeSTiNg est maintenant intégré directement dans INET. Cette intégration modifie le format des modèles d'entrée du simulateur et n'est pas pas pris en charge par les contributions de génération de modèles de simulation qui seront présentées dans le chapitre 8. En effet, la démonstration de la possibilité de générer des modèles de simulation pour plusieurs simulateurs est démontrée dans cette thèse qui n'a pour autant pas vocation à suivre chaque évolution de ces outils, ce qui pourra être fait ultérieurement.

D'autres outils de simulation ne reposant pas sur OMNeT++ existent, comme RTaW-Pegase. RTaW-Pegase est un produit commercial qui supporte davantage de standards que les outils précédemment présentés et qui permet de simuler et d'effectuer des analyses sur des réseaux TSN. Les analyses proposées par cet outil concernent la latence de bout en bout des flux de données dans le pire cas.

Les simulateurs réseaux ne sont pas le seul type d'outil pouvant être utilisés pour assister la conception et la configuration de réseaux TSN. Les travaux présentés dans [GHS+21] et dans [LZW+21] utilise UPPAAL [BLL+96], un outil de vérification de modèles utilisant des automates temporisés [AD94].

Dans les travaux présentés dans [LZW+21], UPPAAL est utilisé pour modéliser et analyser les différents mécanismes d'ordonnancement de trafic des standards TSN, comme le CBS. Dans [GHS+21], UPPAAL est utilisé pour définir un modèle formel permettant d'analyser l'utilisation du TAS. L'utilisation du mécanisme de préemption de trames conjointement avec ces mécanismes d'ordonnancement de trafic est également analysée.

D'autres travaux existent qui ne permettent pas d'assister l'utilisateur dans la vérification du comportement d'un réseau déjà configuré, mais qui assiste l'étape de configuration directement. Ces travaux incluent par exemple des synthétiseurs de configuration pour des mécanismes

10. https ://omnetpp.org/
11. https ://inet.omnetpp.org/

d'ordonnancement de trafic. Un de ces outils est TSNsched [SSN19], qui utilise un solveur de problèmes SMT (*Satisfiability Modulo Theories*) : z3 [dMB08]. TSNsched permet de synthétiser la configuration du TAS. Grâce à un systèmes de contraintes permettant d'encadrer le problème d'ordonnancement, la configuration produite, s'il est possible d'en synthétiser une, permettra de respecter les exigences de latences de bout en bout et de gigue spécifiées pour chaque flux de données.

Les travaux effectués dans [COCS16] présentent d'autres méthodes de synthèse de configuration pour le TAS. Une autre approche est utilisée dans [PO18], qui synthétise la configuration du TAS à l'aide d'un algorithme génétique plutôt que d'un solveur SMT. Dans ces travaux, l'approche basée sur un algorithme génétique permet de produire une solution à la fois pour le problème d'ordonnancement de trafic avec le TAS et pour le problème de routage des flux de données.

Une approche de génération automatique de configuration est présentée dans [HBA+21]. L'outil proposé dans ces travaux permet à l'utilisateur de configurer automatiquement la simulation d'un réseau TSN par NeSTiNg. Cet outil est intégré à OMNeT++ comme un module externe et ne peut donc être utilisé que pour cet environnement de simulation.

3.4 Conclusion

Dans ce chapitre, nous avons tout d'abord présenté une définition de ce que nous appelons la modélisation. Nous avons ensuite présenté un historique des approches de modélisations logicielle. Dans le cadre de notre approche, nous utilisons le standard UCM afin de modéliser l'architecture logicielle.

Ce standard, et l'outil qui l'accompagne, Sigil-UCM, nous permettent de créer une représentation de l'application qui utilisera le réseau que nous concevons. Cette application forme, avec la topologie du réseau, le système et c'est pourquoi notre approche a pour objectif de lier les deux approches de modélisation réseau et logicielle.

Enfin, nous avons présenté les différents outils qui sont utilisés lors de la conception d'un réseau TSN, principalement des simulateurs réseau. Notre approche vise à permettre l'utilisation de plusieurs de ces outils car ils n'ont pas tous les mêmes fonctionnalités et ils ne supportent pas tous les mêmes standards TSN. De plus, l'utilisation de plusieurs outils permet la comparaison des résultats obtenus. Nous avons donc sélectionné plusieurs de ces outils : Mininet, NeSTiNg et RTaW-Pegase.

Les deux simulateurs réseau NeSTiNg et RTaW-Pegase nous permettent de couvrir une large part des standards TSN et plus particulièrement ceux qui nous intéressent le plus : les deux mécanismes d'ordonnancement TAS et CBS.

Néanmoins, l'utilisation de ces outils pour vérifier une configuration du réseau n'est pas suffisante en elle-même. En effet, la modélisation du réseau et de ses flux de données n'est pas une étape simple de la conception. Pour que les résultats obtenus grâce aux simulations du réseau nous permettent de valider une configuration, il est nécessaire de s'assurer que le modèle de simulation représente correctement le système étudié. La problématique qui suit la modélisation est donc celle de la cohérence entre le modèle produit et le comportement réel du système.

C'est une des raisons de notre choix d'UCM pour la modélisation logicielle : il est possible dans l'outil Sigil-UCM de spécifier le comportement des composants de l'application, à partir duquel les flux de données doivent être modélisés, et de produire le code technique de l'application, qui définit le comportement réel de l'application. La cohérence entre les deux est assurée

dans notre approche par la production automatique du modèle des flux de données à partir du modèle de l'architecture logicielle. Cette contribution sera présentée dans le chapitre 6.

Deuxième partie

Contributions

Chapitre 4

Problématiques et approche

L'objectif de cette thèse est de proposer une approche allant de l'identification des éléments à prendre en compte pour le paramétrage d'un réseau TSN (*Time-Sensitive Networking*) à la configuration effective de ce réseau. Plusieurs obstacles, présents à chaque étape de la conception, doivent être franchis pour atteindre ce but.

La complexité inhérente à TSN, de part le nombre et la diversité des standards qui le composent, est le premier de ces obstacles et demande donc un moyen de la maitriser, ce que nous proposons de faire par l'utilisation de modèles.

Assurer la cohérence entre le système modélisé – et donc pour lequel la configuration du réseau est produite – et le système réel qui sera déployé réclame, en plus de la modélisation du réseau, un moyen d'unifier la conception du réseau et celle des applications qui l'utiliseront. Notre approche consiste à modifier à la fois la topologie du réseau, les flux de données qui seront échangés et les applications réparties qui seront déployées sur le réseau et qui produiront ces flux.

La vérification de la validité de la configuration produite – c'est-à-dire de sa capacité à respecter les exigences définies dans le cahier des charges – nécessite l'utilisation d'outils d'analyse et de simulation. Du fait du caractère récent des standards TSN, ces outils n'implémentent pas l'ensemble des nouvelles fonctionnalités. Une utilisation combinée de plusieurs de ces outils devient donc nécessaire, ce qui permet également la comparaison des résultats.

Afin d'illustrer ces différents problèmes, nous utiliserons un cas d'exemple simple basé sur un cas d'étude de drone. Ce système est volontairement simplifié et librement inspiré d'une architecture logicielle de drone volant. Le système embarqué a pour objectif de suivre les consignes de destination qui peuvent être transmises depuis le sol par le biais d'une liaison sans-fil.

4.1 Problèmatiques

Notre travail traite trois problématiques principales. Tout d'abord, nous proposons une façon de modéliser le problème à résoudre. Une telle modélisation vise à caractériser la topologie du réseau et les flux de données qui sont échangés sur celui-ci.

Nous poursuivons la démarche en proposant une façon de calculer automatiquement le modèle des flux de données à partir d'une modélisation des applications de contrôle-commande qui seront déployées sur le réseau.

Enfin, nous implémentons un outil permettant de produire des modèles de simulation pour différents outils de simulation et d'analyse réseau ainsi que des fichiers de configuration pour du matériel réseau. L'objectif est de pouvoir combiner plusieurs vérifications de façon cohérente

afin de s'assurer de la validité de la configuration du réseau.

Le périmètre de notre travail couvre partiellement le calcul de la configuration des mécanismes d'ordonnancement de TSN. Une partie de ce calcul, qui constitue une problématique à part entière, est assurée par d'autres travaux, comme [SSN19] ou [COCS16]. Toutefois, nous proposons une contribution à la façon de calculer cette configuration dans le cas de l'utilisation conjointe de deux mécanismes d'ordonnancement de TSN, utilisés pour des flux de criticité mixte de type contrôle-commande et multimédia.

4.1.1 Modélisation

TSN est un ensemble de standards publié par le groupe de travail IEEE 802.1 TSN qui ajoute de nouvelles fonctionnalités à Ethernet.

Il permet la transmission de flux de données ayant des niveaux de criticité mixte. Les flux de données critiques et non critiques peuvent se partager un même réseau tout en maintenant la garantie que le trafic non critique ne va pas gêner le trafic critique et lui faire rater ses échéances.

En revanche, l'utilisation des nouvelles fonctionnalités introduites par ces standards a un coût : l'augmentation de la complexité de conception du réseau en termes de configuration, de simulation et de vérification et validation.

La première raison pour cette augmentation de complexité est le grand nombre de nouvelles fonctionnalités et la diversité des différents aspects du réseau qu'elles affectent : l'ordonnancement du trafic, la redondance des transmissions, la sécurité, etc.

La nature « à la carte » de TSN participe à cette augmentation de complexité. Il est possible de sélectionner les standards TSN qui seront utilisés en fonction des besoins de chaque système. Par exemple, pour un système ne contenant que des flux de contrôle-commande, on aura tendance à privilégier des mécanismes d'ordonnancement qui minimisent la latence de bout en bout et qui empêchent les flux de se bloquer entre eux. En fonction du besoin de sûreté, des mécanismes de TSN permettant la mise en place d'une redondance par duplication et élimination de trames peuvent également être utilisés. Dans un système plutôt orienté vers la transmission de flux vidéo d'autres mécanismes de TSN seront privilégiés, notamment ceux permettant de répartir l'accès au réseau de façon à ce qu'aucun flux de données ne se retrouve bloqué par d'autres. Un tel mécanisme permet d'empêcher qu'une classe de trafic ne s'arroge une part trop importante de la bande passante disponible en attribuant un proportion de cette bande passante à chaque classe de trafic.

Nous avons identifié quelques-uns des standards TSN comme étant particulièrement intéressants et bien adaptés à un large ensemble de besoins. Ces standards sont :

— IEEE 802.1AS [AS20] : Ce standard définit le protocole gPTP (*Generalized Precision Time Protocol*). Ce protocole permet la synchronisation de chaque nœud du réseau ;

— IEEE 802.1Qav [Q18] : Ce standard définit le mécanisme d'ordonnancement CBS (*Credit-Based Shaper*). Ce mécanisme d'ordonnancement permet de répartir l'accès au réseau de manière à ce qu'une classe de trafic ne puisse pas bloquer indéfiniment les autres ;

— IEEE 802.1Qbv [Q18] : Ce standard définit ce qui permet d'utiliser le mécanisme d'ordonnancement TAS (*Time Aware Shaper*). Ce mécanisme d'ordonnancement permet de diviser temporellement l'accès au réseau de façon à ce que les différentes classes de trafic puisse bénéficier de fenêtre temporelles pendant lesquelles les trames qui leur sont associées peuvent être transmises sans être gênées par d'autres transmissions.

Dans le cadre de nos travaux, nous considérons que les flux de type contrôle-commande, ayant des contraintes temps réel dures et le niveau de criticité le plus élevé, relèvent du TAS.

Les flux de données de type multimédia, ayant des contraintes temps réel souple ou ferme et un niveau de ciriticité moins élevé, relèvent du CBS.

La complexité de TSN nous mène à la conclusion qu'il est nécessaire de disposer d'un moyen de formellement définir le problème de conception à résoudre pour chaque nouveau système. Un moyen d'obtenir cette formalisation est de définir les concepts permettant de créer un modèle du problème à résoudre. Ce modèle correspond à une représentation de chaque élément du réseau. Une contribution de cette thèse est donc de définir formellement les concepts permettant la création d'un modèle de réseau TSN avec des liaisons filaires. La création des modèles est finalement la responsabilité des architectes système et réseau.

Dans l'exemple de la figure 4.1 que nous considérons dans ce chapitre, la cahier des charges établit un ensemble de contraintes qui devront être respectées lors de l'étape de modélisation.

Cet exemple est un système supposé être embarqué sur un drone ayant la capacité de recevoir des consignes de destination pendant son vol par le biais de communications sans-fil et ses contraintes sont les suivantes :

— une boucle de contrôle-commande, avec une période de 30 ms, doit être maintenue entre le composant contrôlant le moteur du drone et le moteur lui-même. Ce flux est critique, son échéance est fixée à 10 ms et sa gigue doit être minimisée ;

— des composants permettant d'obtenir des informations de position et de cap – une boussole et un GPS – transmettent ces informations au composant gérant la navigation avec une période de 90 ms ;

— les nouvelles consignes de destination ne peuvent pas être reçues plus d'une fois toutes les 180 ms ;

— la réception des nouvelles consignes de destination doit être assurée par un composant séparé du reste, afin d'être proche de l'antenne ;

— du fait des contraintes d'espace à bord du drone, les différents terminaux du système ne peuvent pas être tous regroupés en un point du véhicule. Ils doivent être répartis et l'utilisation de 2 ponts afin de les relier est nécessaire. Cette répartition des nœuds de calcul est également faite par soucis de modularité de l'architecture système.

En suivant ces contraintes, l'étape de modélisation fixe la topologie du réseau, présentée dans la figure 4.1. Le réseau contient 4 terminaux possédant 1 port chacun, 2 ponts possédants 3 ports chacun. Le flux de données contenant les nouvelles consignes de destination sera émis par le terminal *Passerelle*, gérant les communications sans-fil, et sera transmit au terminal *Mission*, gérant la navigation, en passant par les ponts *Pont1* et *Pont2*. Après avoir pris en compte les informations de position et de cap, le terminal *Mission* transmettra une nouvelle consigne au composant gérant le moteur. Ce flux sera transmis dans tous les cas – même s'il n'y a pas de nouvelle consigne de destination – avec une période de 30 ms. La boucle de flux de données de contrôle-commande sera émise d'abord par le terminal *Controle*, contrôlant le moteur, et reçue par *Moteur*, ce qui déclenchera la réponse de *Moteur* à *Controle* par le chemin inverse, transmettant l'état actuel du moteur.

Les bandes passantes des équipements réseau sont de 1 Gbps et les délais de traitement des ponts sont de 1 µs. Les temps de propagation sont considérés comme négligeables étant donné la faible longueur des liens. Pour compléter le modèle, il reste à fournir le modèle des flux de données, en termes de taille des données, de période et de phase de transmission etc.

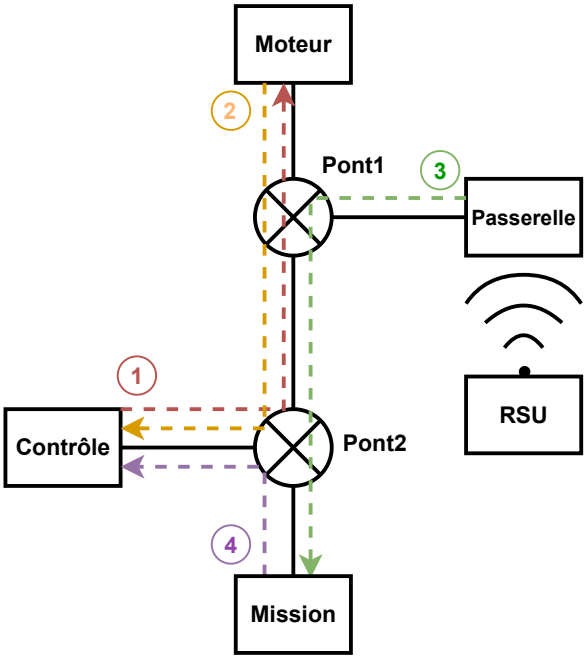


FIGURE 4.1 – Topologie du réseau embarqué à bord du drone.

4.1.2 Calcul automatique des modèles des flux de données

Bien qu'il soit tout à fait possible pour l'architecte système de formuler un cahier des charges contenant les exigences du système en termes d'équipements réseau, de contraintes de latence et de gigue sur un ensemble de flux de données et de s'appuyer dessus pour créer un modèle du réseau, un problème se pose lors de cette étape : comment s'assurer que les valeurs des paramètres du modèle sont correctes et cohérentes avec la réalité du système une fois déployé ?

Ce problème fait apparaître un nouveau besoin de cohérence lors de l'étape de modélisation du réseau. La solution proposée par cette thèse est de ne pas se limiter à la simple modélisation du réseau. Le but de la modélisation du réseau est de définir entièrement le problème à résoudre de façon à pouvoir en calculer une solution qui prendra la forme d'une configuration du réseau. Les paramètres ayant le plus grand impact sur la production de cette configuration sont ceux liés aux flux de données, or ces paramètres dépendent avant tout du fonctionnement des applications qui utiliseront le réseau.

Nous proposons donc de combiner l'utilisation de la modélisation du réseau avec la modélisation des applications qui l'utiliseront. Le modèle des applications doit permettre deux choses :

— le calcul des paramètres des flux de données (e.g. période et phase de transmission, volume de données, etc.) ;

— la production d'au moins une partie du code source des applications permettant d'assurer la cohérence entre les paramètres des flux calculés et le comportement réel de l'application déployée.

Dans ces conditions, un unique modèle des applications permet d'assurer la cohérence entre la partie du modèle du réseau la plus cruciale – et la plus difficile à obtenir – et le fonctionnement réel des applications afin de garantir que la configuration réseau produite permet bien de répondre au problème de conception d'un système formellement définit.

L'approche de modélisation des applications que nous avons sélectionnée pour ces travaux est UCM [UCM21] et son implémentation dans l'outil Sigil-UCM [Ver23]. Le standard UCM est une application des principes du CBSE (*Component-Based Software Engineering*). Le standard, et les fonctionnalités que nous utilisons sont présentés dans le chapitre 3 et dans l'annexe B.

Les composants logiciels, modélisés en UCM, de type *Controle* et *Moteur*, présentés dans la section 3.2, sont déployé sur les nœuds du même nom que sur la topologie présentée dans la figure 4.1.

Dans le cas de notre exemple, le modèle de l'architecture logicielle spécifie le comportement des composants responsables des transmissions des flux de contrôle-commande. Cette spécification indique que le flux de données émis par le terminal *Moteur* vers *Controle* est déclenché par la réception du flux de données émis par *Controle* vers *Moteur*. La spécification du comportement des composants sera abordée en détail dans le chapitre 6.

La phase de transmission du flux de données allant de *Moteur* à *Controle* dépend donc du temps total de transmission du flux allant de *Controle* à *Moteur*. Grâce à la spécification du comportement de ces applications, il est possible de calculer cette phase et, de la même façon, de garantir que le modèle des flux de données correspond à la réalité du comportement des applications, puisque leur code est également généré à partir de cette même spécification.

4.1.3 Combinaison du TAS et du CBS

Le système que nous utilisons comme exemple ne contient actuellement qu'un flux acheminant les nouvelles consignes et une boucle de deux flux de contrôle-commande. Ce système étant embarqué à bord d'un véhicule, il est tout à fait envisageable de vouloir le faire évoluer, en tirant parti des capacités de TSN, pour qu'il permette aussi la transmission de flux de données non critiques, comme des flux multimédia (audio et vidéo).

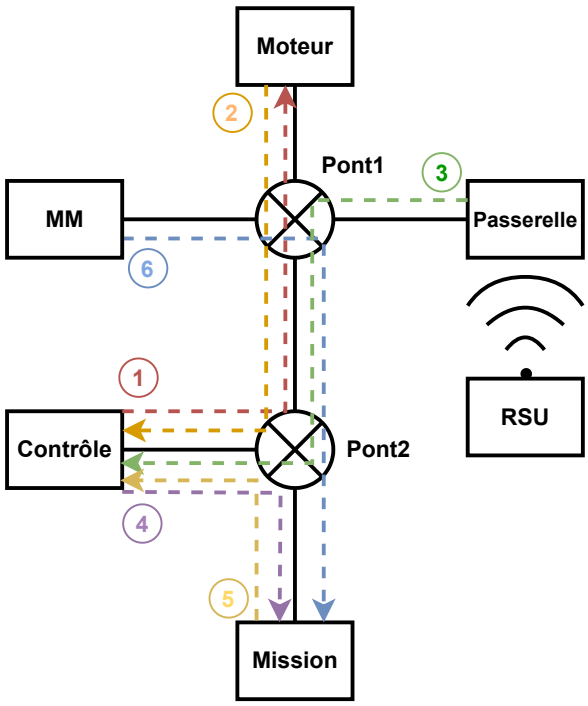


FIGURE 4.2 – Évolution de la topologie du réseau embarqué dans le véhicule, 2 terminaux sont ajoutés pour la transmission de flux multimédia.

Nous pouvons donc faire évoluer le réseau et sa topologie pour ajouter des terminaux produisant et recevant des flux vidéo, comme présenté par la figure 4.2. Le terminal MM produit des flux vidéo à destination de *Mission*, dont le rôle sera celui d'un tableau de bord, ces flux seront abordés dans le détail dans le chapitre 7.

La présence sur le réseau de flux d'une nature différente de ceux existants précédemment peut conduire à une modification des mécanismes d'ordonnancement utilisés. En l'occurrence, le TAS, utilisé pour les flux de contrôle-commande, n'est pas le plus indiqué pour l'ordonnancement de flux vidéo ; on lui préfère généralement le CBS. L'évolution du système menant à la transmission de ces différents flux de données va nous conduire à utiliser différents mécanismes d'ordonnancement conjointement.

La complexité de TSN peut être encore davantage mise en évidence par ces deux mécanismes d'ordonnancement de trafic. En effet, ils nécessitent leur propre configuration à chaque port de sortie utilisé par les flux de données dont ils gèrent l'ordonnancement. Cette étape représente un important effort de configuration car obtenir un ordonnancement du trafic valide, c'est-à-dire permettant de respecter les échéances de tout les flux de données concernés. La complexité du problème de configuration de ces deux mécanismes est encore augmentée lorsqu'ils sont utilisés conjointement, l'utilisation du TAS ayant un impact sur le fonctionnement du CBS.

Grâce à notre approche de modélisation, l'évolution du système est gérée simplement, par l'ajout aux modèles du réseau et des applications des nouveaux composants. Ces changements seront répercutés sur l'ensemble de ce qui est généré automatiquement à partir de ces modèles : le code des applications, le modèle des flux de données, la configuration du réseau, des modèles d'analyse et de simulation et la documentation du réseau et des applications.

Nous proposons donc une contribution permettant de calculer la configuration du CBS prenant en compte le fait qu'il est utilisé conjointement au TAS.

4.1.4 Génération des modèles de simulation et des fichiers de configuration

Une fois la configuration du réseau obtenue, une dernière étape reste encore à franchir avant le déploiement du système : la vérification de la configuration.

Une autre raison à l'augmentation de la complexité de conception est l'impact qu'a TSN sur les outils utilisés pendant cette phase. À notre connaissance, les outils utilisés pour simuler et analyser des réseaux TSN lors de leur conception ne supportent qu'un sous-ensemble des nouvelles fonctionnalités apportées par TSN. Certains outils commerciaux semblent supporter davantage de fonctionnalités différentes mais l'utilisation de plusieurs outils aux capacités différentes peut être une nécessité lors de la conception d'un réseau TSN.

De part la complexité de ces réseaux, la fidélité de la simulation et la précision de l'analyse sont des problèmes difficiles et la possibilité de comparer les résultats obtenus par différents outils permet d'augmenter le niveau de confiance dans la configuration produite.

En revanche, chacun de ces outils a sa propre façon de définir et de configurer un réseau TSN, ce qui augmente aussi la complexité de la phase de conception car chaque outil doit être maitrisé.

Synthétiser automatiquement la configuration d'un réseau TSN permet de grandement réduire le risque d'erreur humaine dans l'utilisation d'outils de conception, supprime la nécessité de savoir configurer ces outils et savoir extraire et interpréter les résultats reste la seule nécessité. Le fait de ne plus avoir à manuellement configurer chaque outil représente un gain de temps considérable et, surtout, permet d'assurer la cohérence entre le modèle du réseau et la configuration produite pour chaque outil de conception.

Afin d'illustrer notre approche, nous avons implémenté un outil de synthèse de configuration

et de génération de modèles de simulation à destination de plusieurs outils de conception. Cet outil a été nommé MoBACT (*Model-Based Automatic Configuration for TSN*). Dans son état actuel, MoBACT à la capacité de prendre comme entrée un modèle de réseau TSN, de générer la configuration du TAS et de produire les fichiers de configuration utilisés par trois outils de conceptions : Mininet [12] [LHM10], NeSTiNg [13] [FHC+19] et RTaW-Pegase [14]. Chacun de ces outils a ses propres spécificités.

4.2 Organisation générale des contributions

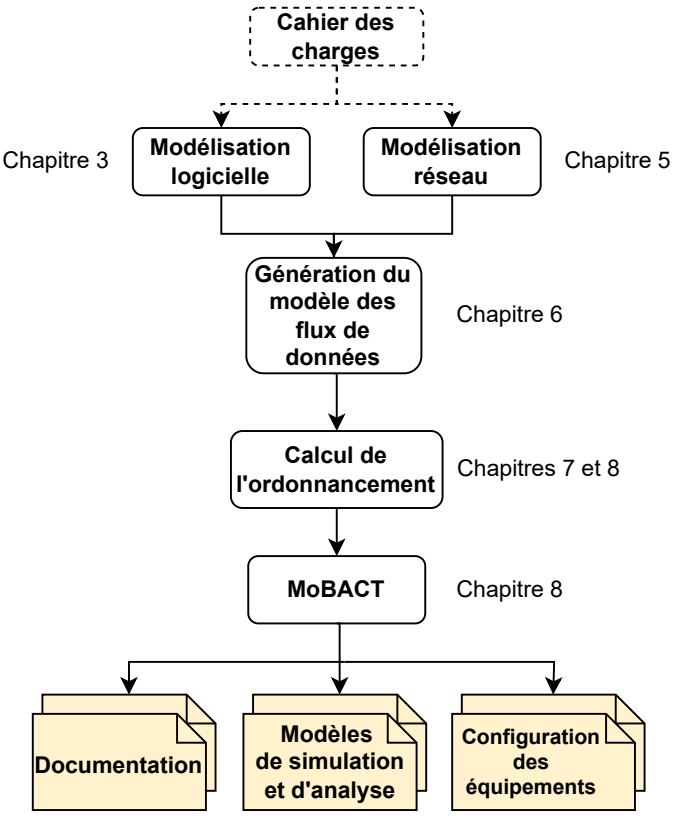


FIGURE 4.3 – Schéma représentant les différentes étapes de notre approche.

Afin d'assister le processus de conception de réseaux TSN et de surmonter la complexité de leur configuration, nous proposons une approche en plusieurs étapes basée sur l'utilisation de modèles, présentée dans la figure 4.3.

L'approche que nous présentons se base sur l'utilisation de modèles et contribue à résoudre les deux problèmes suivants : comment modéliser formellement des réseaux TSN de façon à pouvoir synthétiser automatiquement une configuration et comment l'utilisateur peut bénéficier de l'utilisation de plusieurs outils de conception, comme des simulateurs, sans avoir à subir la courbe d'apprentissage de chacun d'entre eux et tout en limitant au maximum le risque d'incohérences entre les modèles de simulation et le comportement réel du système.

12. http://mininet.org
13. https://gitlab.com/ipvs/nesting
14. https://www.realtimeatwork.com/rtaw-pegase/

Le point de départ de notre approche, à partir duquel les contributions que nous proposons sont appliquées, est la définition du cahier des charges du système, qui contient les différentes exigences que le système doit être capable de respecter ; ces exigences n'ont pas besoin de suivre un format prédéfini, elle serviront à créer le modèle lors de l'étape suivante. Les exigences peuvent concerner plusieurs aspects du réseau : le nombre de ponts, le nombre de terminaux, la topologie du réseau, les communications critiques et leurs échéances, etc. Dans le cas d'un réseau utilisé dans un drone, par exemple, les contraintes de poids et d'espace auront une grande influence sur le choix des équipements, de leur nombre et de leur disposition.

La première étape de notre approche est la modélisation. L'objectif de cette étape est de créer une représentation de chaque élément du réseau en se basant sur le cahier des charges établi lors de l'étape précédente. Ce modèle doit pouvoir contenir toutes les informations nécessaires à la synthèse de configuration et à la génération des fichiers utilisés par l'ensemble des outils de conception cibles. Une fois complété, ce modèle permet d'assurer la cohérence entre les configurations générées pour chaque outil de conception puisqu'il est la seule référence. Les contributions utilisées pour cette étape sont présentées dans les chapitres 5 et 6.

La deuxième étape de notre approche est le calcul de la configuration des mécanismes d'ordonnancement de TSN. L'objectif de cette étape est de compléter le modèle avec cette configuration, en la calculant de façon à ce que les contraintes du système soient respectées.

Le principal intérêt de cette étape est donc la synthèse de la configuration du TAS et du CBS en se basant sur l'ensemble des données présentes dans le modèle : topologie, chemins des flux de données, taille des données transmises, échéances des flux de données, etc. Les GCL de chaque port de sortie concerné par la transmission d'un flux de données appartenant à la classe de trafic contrôle-commande sont donc synthétisées et permettront de garantir le respect de leurs échéances. Ce calcul est réalisé par un outil produit dans le cadre des travaux présentés dans [SSN19] et dont le fonctionnement est présenté dans le chapitre 8. Notre contribution, présentée dans le chapitre 7, permet, en s'appuyant sur les mêmes éléments, de calculer la configuration du CBS à utiliser à chaque port de sortie concerné par la transmission d'un flux de données appartenant à la classe de trafic multimédia.

Enfin, une fois le modèle complété avec cette configuration, un ensemble de fichiers est automatiquement généré, principalement à destination des outils de conception cibles. Cette étape de génération permet aussi de produire un ensemble de fichiers de documentation sur le réseau, réunissant toutes les données présentes dans le modèle dans un format facilement consultable. Des fichiers de configuration contenant le résultat de la synthèse de configuration du TAS et du CBS sont également produit à destination d'équipements réseau réels. L'outil que nous avons développé pour cette étape, MoBACT (*Model-Based Automatic Configuration for TSN*) est présenté dans le chapitre 8.

Il est possible de facilement itérer entre les différentes étapes de notre approche, présentés dans la figure 4.3 en créant une première version du modèle du réseau et en la testant puis en faisant évoluer le modèle du réseau. En fonction des simulations permises par l'étape de génération, il est possible à l'utilisateur de modifier la topologie du réseau et la configuration de l'application et relancer un calcul de la configuration puis des simulations pour vérifier la nouvelle architecture. Ce processus itératif est grandement facilité par l'utilisation d'un modèle du réseau, qui centralise tous les changements à apporter en un seul et même endroit et assure qu'ils seront répercutés sur l'ensemble des fichiers produits pour chaque outil de conception.

Chapitre 5

Modélisation d'un réseau TSN

L'étape de modélisation est la base de notre processus de conception de réseau TSN ; c'est sur elle que repose l'ensemble de notre approche : la garantie de cohérence entre l'ensemble des modèles de simulations et d'analyse produits, le gain de temps lors des modifications du système, l'économie du temps d'apprentissage des différents outils et la formalisation du problème à résoudre.

La garantie de cohérence est apportée par le fait de n'utiliser qu'un seul et unique modèle central. Ce modèle est donc utilisé comme référence pendant la totalité du processus de conception du réseau et toute génération sera faite à partir de celui-ci. Lorsqu'il est nécessaire de faire évoluer le système, ce modèle central est le seul endroit où les modifications doivent être apportées. Il ne peut donc pas y avoir de perte de cohérence entre différents modèles utilisés à différentes étapes du processus de conception.

Ce chapitre fait la liste des paramètres qu'il est nécessaire de fournir pour modéliser un réseau TSN de façon à pouvoir synthétiser une configuration des mécanismes d'ordonnancement et générer un ensemble de modèles de simulation et d'analyse pour différents outils. Cet ensemble de paramètres constitue les ressources de modélisation, autrement dit, la définition des concepts qui seront utilisés pour représenter chaque élément d'un réseau TSN dans un modèle.

La définition de ces ressources de modélisation a été inspirée par le standard MARTE [MAR19], utilisé dans le domaine des systèmes temps réels embarqués et qui nous sert de base. Ce standard propose deux concepts – les ressources de communication et les ressources de calcul – que nous réutilisons pour définir l'ensemble des ressources de modélisation nécessaires pour un réseau TSN, c'est-à-dire : les terminaux, les ponts, les liens et les flux de données.

L'instanciation de ces définitions, en donnant des valeurs à chaque paramètre permet la création de modèle de réseaux TSN. Le modèle des ressources ainsi que les modèles de réseaux TSN créés à partir de ce dernier utilisent une syntaxe XML.

5.1 Formalisation de la modélisation de réseau TSN

Nous modélisons un réseau TSN comme un ensemble de nœuds représentant les terminaux et les ponts, reliés entre eux par des liens Ethernet et sur lequel un ensemble de flux de données circule, auxquels s'appliquent des exigences venant du système.

Définition 5.1.1. (Réseau). Soit R un réseau TSN, R est défini par $R :=< Nœud, Lien, Flux, Exigence >$ avec :

— *Nœud* est l'ensemble des nœuds du réseau et représente aussi bien les terminaux que les ponts ;

— *Lien* est l'ensemble des liens Ethernet qui relient les nœuds entre eux ;

— *Flux* est l'ensemble des flux de données qui circulent sur le réseau ;

— *Exigence* est l'ensemble des exigences système qui s'appliquent aux flux de données.

Définition 5.1.2. (Nœud). Soit un réseau R et $n \in R.Nœud$ un nœud qui représente un terminal ou un pont, n est caractérisé par l'ensemble de ses interfaces Ethernet : $n :=< Interfaces >$. Dans le cas d'un pont, le nœud est également caractérisé par un paramètres $R_{commutation}$ correspondant au délai de traitement du pont.

Définition 5.1.3. (Interface). Soit un nœud $n \in R.Nœud$ et une interface Ethernet $i \in n.Interfaces$, i est caractérisée par sa bande passante. Dans le cas d'une interface appartenant à un terminal, elle est également caractérisée par ses adresses MAC et IP : $i :=< BW, A_{MAC}, A_{IP} >$ avec :

— BW la bande passante disponible à l'interface i ;

— A_{MAC} l'adresse MAC de l'interface de terminal i ;

— A_{IP} l'adresse IP de l'interface de terminal i.

Définition 5.1.4. (Lien). Soit un réseau R et $l \in R.Lien$ un lien Ethernet du réseau reliant deux nœuds entre eux, l est caractérisé par le couple de ses extrémités et par un délai de propagation : $l :=< i_1, i_2, R_{propagation} >$ avec :

— i_1 et i_2 sont les deux interfaces aux extrémités de l ;

— $R_{propagation}$ est le délai de propagation présent sur le lien (il est proportionnel à la longueur du câble mais est généralement négligeable dans le cas des systèmes embarqués dont les liens sont de petites longueurs).

Définition 5.1.5. (Flux). Soit un réseau R et $f \in R.Flux$ un flux de données circulant sur le réseau, f est caractérisé par une interface Ethernet émettrice, une interface Ethernet destinataire, une charge utile, un niveau de priorité, une phase et un éventuel moment d'arrêt de transmission : $f :=< src, dest, pl, pcp, \Phi, end >$ avec :

— *src* l'émetteur du flux de données (aussi appelé *talker*) ;

— *dest* le destinataire du flux de données (aussi appelé *listener*) ;

— *pl* la charge utile transmise par ce flux de données en octets ;

— *pcp* le niveau de priorité du flux de données parmi les huit niveaux possible numérotés de 0 pour le moins prioritaire à 7 pour le plus prioritaire ;

— Φ la phase de transmission du flux de données par rapport au démarrage du système ;

— *end* le moment auquel le flux de données arrête sa transmission, ce paramètre n'est pas obligatoire.

Définition 5.1.6. (Flux périodique). Un flux périodique est un flux de données auquel est associé une période T. La période indique le temps entre deux émissions d'un flux de données périodique.

Définition 5.1.7. (Flux sporadique). Un flux sporadique est un flux de données auquel est associé un intervalle minimal d'interarrivés R_{inter}. Cet intervalle indique le moment le plus tôt auquel un flux de données sporadique peut être émis à nouveau.

Définition 5.1.8. (Exigence). Soit un réseau R et $e \in R.Exigence$ une exigence système. Une exigence système s'applique à au moins un flux de données et est caractérisée par la liste des flux de données qui lui sont associés, une contrainte de latence de bout en bout, une éventuelle contrainte de gigue maximale, l'identifiant du VLAN utilisé, une indication de la criticité temporelle et par les chemins des flux de données : $f :=< F_e, D, J, vlan, C >$ avec :

— F_e un sous-ensemble des flux de données auxquels s'applique l'exigence e ($F_e \subseteq R.Flux$) ;
— D l'échéance, c'est-à-dire la latence de bout en bout, que ne doit pas dépasser le flux de données ;
— J la valeur de la gigue que le flux de données ne doit pas dépasser (ce paramètre n'est pas obligatoire) ;
— *vlan* l'identifiant du VLAN utilisé par les flux de données auxquels s'applique l'exigence e ;
— C la liste des chemins que peuvent emprunter les flux de données F_e.

Définition 5.1.9. (Chemin). Un chemin est une suite d'interface Ethernet reliant l'émetteur d'un flux de données à son destinataire.

Les interfaces sont les éléments du modèle qui vont porter la configuration des mécanismes d'ordonnancement. C'est donc à ces éléments que sont associées les configurations du TAS et du CBS.

Définition 5.1.10. (Configuration TAS). Soit un réseau R et un nœud $n \in R.Nœud$. Une interface $i \in n.Interface$ peut posséder une configuration du TAS qui est caractérisée par la longueur de son cycle et par une liste de fenêtres temporelles : $C_{TAS} :=< L_{TAS},\ TS >$ avec :

— L_{TAS} la longueur du cycle TAS qui se répètera et sera divisé en fenêtres temporelles ;
— TS la liste non vide de fenêtres temporelles qui composent le cycle.

Définition 5.1.11. (Fenêtre temporelle). Soit une configuration TAS C_{TAS} et une fenêtre temporelle $ts_i \in C_{TAS}.TS$. Une fenêtre temporelle est caractérisée par sa durée et par la liste des niveaux de priorités desquels les flux de données associés pourront être transmis pendant la durée de la fenêtre : $ts_i :=< L_{ts},\ PCP >$ avec :

— L_{ts} la durée de la fenêtre temporelle ;
— PCP la liste des niveaux de priorités autorisés à être transmis.

Définition 5.1.12. (Configuration CBS). Soit un réseau R et un nœud $n \in R.Nœud$. Une interface $i \in n.Interface$ peut posséder une configuration du CBS par niveau de priorité (au nombre de huit). Ces configurations du CBS sont caractérisées par une proportion de la bande passante disponible à l'interface i et par le niveau de priorité concerné : $C_{CBS} :=< \alpha^+,\ pcp >$ avec :

— α^+ la proportion de la bande passante allouée à un niveau de priorité (appelé *idle slope* en anglais) ;
— *pcp* le niveau de priorité concerné par cette configuration du CBS.

5.2 Besoin que le modèle doit satisfaire

Afin d'atteindre notre objectif de pouvoir utiliser un modèle du réseau central, permettant la génération de la configuration du réseau et des modèles de simulation à partir de cette unique représentation du réseau, il est nécessaire de définir les concepts dont nous avons besoin et les paramètres qui leur sont associés. Ces concepts serviront à créer les représentations de chaque éléments du réseau et il est donc indispensable que leurs paramètres soient définis de manière à permettre la génération de configuration et de modèles de simulation pour différents outils.

Dans cette section, les diagrammes de classe respecteront le standard UML [UML17].

5.2.1 Caractérisation de la topologie réseau

Une topologie réseau inclut des terminaux et des ponts, ayant tous au moins un port, qui sont reliés entre eux par des liens. Un terminal [15] est équivalent à une machine effectuant des calculs et ayant besoin de pouvoir communiquer avec d'autres machines.

Dans notre approche, la topologie est considérée comme une contrainte venant du cahier des charges exprimées au commencement de la conception du système.

Nous différencions deux types de ports (aussi appelés interfaces Ethernet) : ceux des terminaux, ayant une adresse MAC, une adresse IP et une valeur de bande passante, et ceux des ponts, n'ayant qu'une valeur de bande passante.

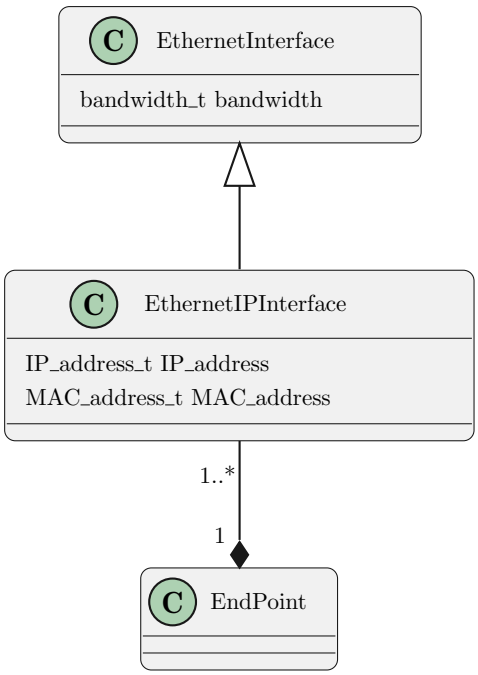


FIGURE 5.1 – Diagramme de classe de la ressource de modélisation servant à modéliser les terminaux.

Le diagramme de classe de la figure 5.1 présente la ressource de modélisation utilisée pour représenter les terminaux. Un ensemble non nul d'interfaces Ethernet compose les terminaux, chaque interface ne peut être associée qu'à un unique terminal. Ces interfaces possèdent une adresse MAC, et une valeur de bande passante qui servira aux calculs de configuration des mécanismes d'ordonnancement. Elles peuvent également posséder une adresse IP dans le cas des interfaces des terminaux. Un terminal n'a donc pour caractéristiques que celles de ses ports.

Le diagramme de classe de la figure 5.2 présente la ressource de modélisation utilisée pour représenter les ponts. Dans le cadre de ces travaux, tous les ponts sont en réalité des ponts possédant des capacités TSN, nous avons fait l'hypothèse que les réseaux conçu à partir de notre approche doivent être purement TSN.

Un pont TSN est caractérisé par un temps de traitement incompressible appelé « latence de commutation » et par une liste des fonctionnalités TSN qu'il supporte, à titre purement indicatif car cette information n'est pas utilisée dans les modèles de simulation générés mais il est néanmoins important de le préciser dans la documentation du réseau.

15. Généralement appelé *end-point* en anglais.

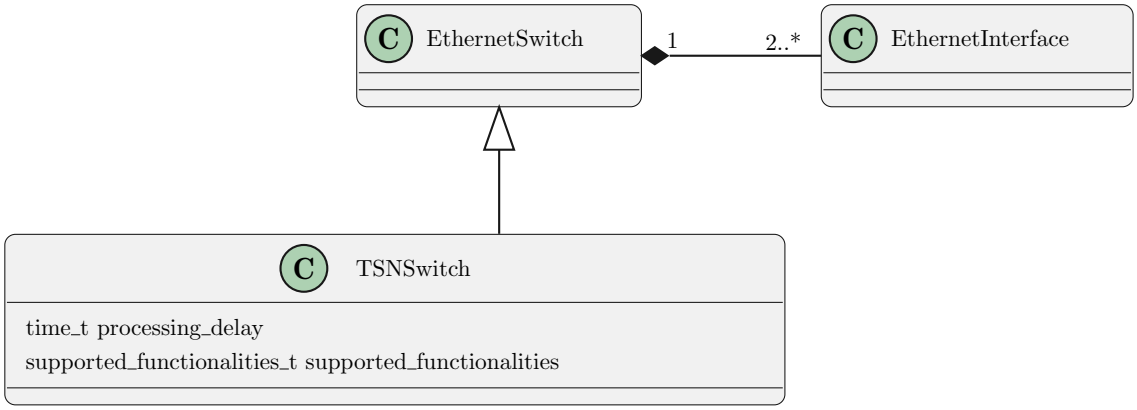


FIGURE 5.2 – Diagramme de classe de la ressource de modélisation servant à modéliser les ponts.

Le rôle des liens Ethernet est de relier les terminaux aux ponts et les ponts entre eux par l'intermédiaire de leurs interface Ethernet.

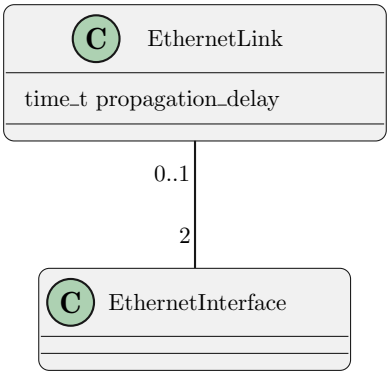


FIGURE 5.3 – Diagramme de classe de la ressource de modélisation servant à modéliser les liens.

Un lien Ethernet, dont la ressource de modélisation est présentée par la figure 5.3, est caractérisé par ses deux extrémités et par un délai de propagation dont la valeur est nulle dans le cas général, car elle est négligeable par rapport au temps de transmission des trames, surtout dans un système embarqué dont la longueur des câbles est faible. Il est toutefois à noter que plus le débit augmente, moins cette valeur est négligeable, par exemple avec un débit de 10 Gb/s le temps de propagation est d'environ 30 ns pour 10 mètres alors que le temps de transmission d'une trame de taille minimale (64 octets) est de 67 ns.

5.2.2 Caractérisation des flux de données

Les flux de données représentent les communications prenant place sur la topologie précédemment définie. La caractérisation des flux de données est séparée en deux parties : leurs caractéristiques intrinsèques et leur allocation sur le système.

Dans le cas des systèmes que nous concevons, ils viennent des applications logicielles de type contrôle-commande, comme abordé dans le chapitre 4, déployées sur les terminaux.

Nous nous focalisons dans le cadre de ces travaux sur deux types de flux de données : les flux

périodiques et les flux sporadiques car ce sont les types de flux de données les plus couramment utilisés pour du contrôle-commande.

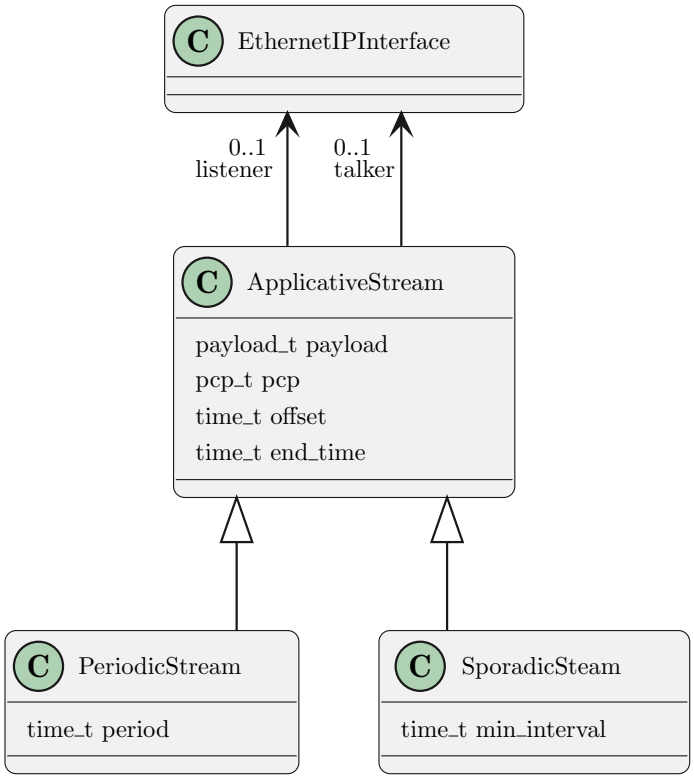


FIGURE 5.4 – Diagramme de classe de la ressource de modélisation servant à modéliser les flux de données.

Le diagramme de classe de la figure 5.4 présente la ressource de modélisation utilisée pour représenter les flux de données. Les flux de données ont un unique émetteur et un unique destinataire, ce choix est discuté dans la section 5.5.

Un flux périodique est caractérisé par la taille de sa charge utile (en nombre d'octets), sa phase et sa période de transmission. D'autres paramètres lui sont également associés : son niveau de priorité (dont la notion est présentée dans le chapitre 2) et un éventuel moment d'arrêt de transmission.

L'heure d'arrêt de transmission peut avoir plusieurs utilisation : un flux de données qui ne sera pas transmis durant toute la durée de vie du système, par exemple pour l'envoie d'une configuration initiale, ou un changement de comportement du système prévu à une date données, par exemple pour la simulation d'un passage en mode dégradé.

Un flux sporadique est caractérisé de la même façon qu'un flux périodique (charge utile, phase, priorité et heure d'arrêt) mais la période est remplacée par un délai minimum de répétition appelé intervalle d'interarrivé.

Les caractéristiques liées à l'allocation des flux de données sur le système sont définis dans une partie séparée du modèle ; ce qui permet de facilement envisager plusieurs scénarios d'allocation sans modifier l'intégralité du modèle. La ressource de modélisation représentant ces exigences système est présentée par le figure 5.5.

Les caractéristiques liées à l'allocation des flux de données sont : leurs exigences de latence et de gigue, comme présenté dans la section 1.4, les chemins empruntés et l'identifiant du VLAN

utilisé. Cet identifiant de VLAN fait partie, tout comme le niveau de priorité, de l'en-tête Ethernet mais ces deux paramètres ne sont pas liés.

Il est également possible de spécifier les émetteurs et les destinataires des flux de données dans cette partie du modèle si cela n'a pas été fait dans les caractéristiques intrinsèques.

Il est possible de prévoir plusieurs chemins pour les flux de données. Ces multiples chemins peuvent désigner les chemins potentiellement empruntables par les flux de données ou, dans le cas de l'utilisation du standard TSN IEEE 802.1CB présenté dans la section 2.2, les différents chemins redondants qui seront empruntés simultanément par les flux de données.

Dans la pratique, cette possibilité prévue par le modèle n'est pas utilisée car ce mécanisme n'est généralement pas supporté par les simulateurs réseaux et il convient donc de ne spécifier qu'un unique chemin pour chaque flux de données. Le modèle est prévu pour anticiper l'évolution de ces outils.

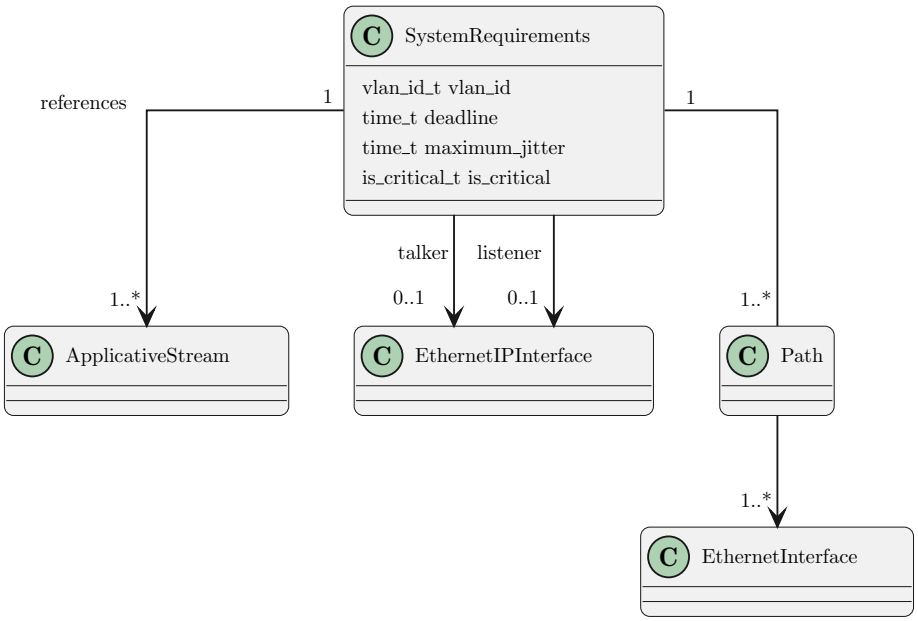


FIGURE 5.5 – Diagramme de classe de la ressource de modélisation servant à modéliser les exigences systèmes qui s'appliquent aux flux de données.

5.2.3 Caractérisation de la configuration des mécanismes d'ordonnancement

La configuration des mécanismes d'ordonnancement est l'ensemble des paramètres spécifiques à chaque mécanisme d'ordonnancement permettant de dicter leur comportement de façon à pouvoir respecter les exigences de latence et de gigue des flux de données.

Il n'existe pas une unique configuration mais une pour chaque mécanisme d'ordonnancement utilisé sur chaque interface du réseau. Par exemple, un réseau composé de deux ponts TSN reliant cinq terminaux, comme présenté dans la figure 5.6 avec les ponts *Pont1* et *Pont2*, et utilisant à la fois le TAS et le CBS utilisera un total de quatorze configurations : une configuration du TAS et une configuration du CBS par port.

La définition de la configuration du TAS, illustrée par la figure 5.7, pour un port donné passe par la spécification d'un cycle par sa durée puis par la spécification de l'ensemble des fenêtres temporelles qui composent ce cycle par leur durée et la liste des niveaux de priorité autorisés à transmettre.

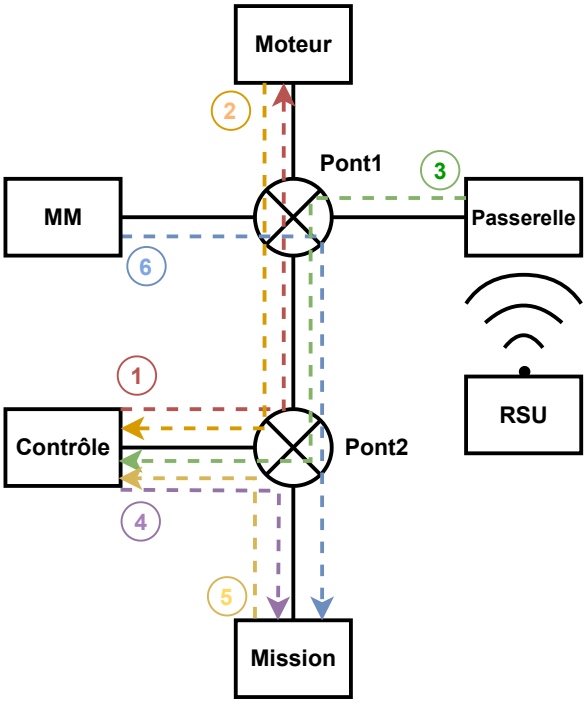


FIGURE 5.6 – Schéma de la topologie du réseau utilisé comme exemple dans le chapitre 4.

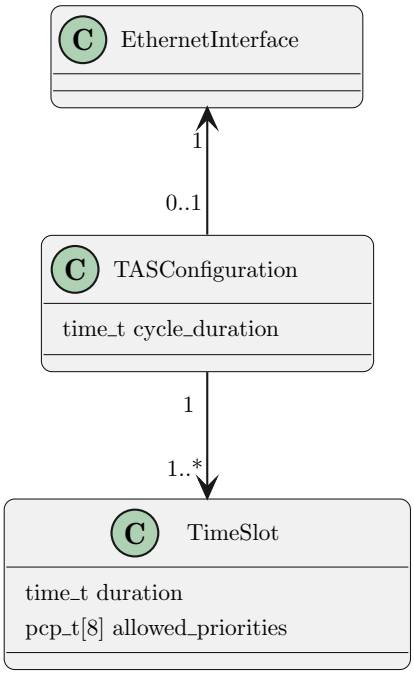


FIGURE 5.7 – Diagramme de classe de la ressource de modélisation servant à modéliser la configuration du TAS.

La configuration du TAS concernant tous les niveaux de priorités, une seule (c'est-à-dire un seul cycle), est définie pour chaque interface. En revanche, il peut y avoir un nombre arbitraire de fenêtres temporelles par cycle.

La définition de la configuration du CBS, illustrée par la figure 5.8, pour un port donné permet à l'utilisateur de définir, pour chaque interface, une proportion de bande passante allouée à un niveau de priorité spécifique. Il est donc possible pour l'utilisateur de spécifier huit valeurs de proportion de bande passante, une par niveau de priorité, à chaque interface. Cet ensemble de valeurs forme la configuration du CBS pour une interface.

La valeur des proportions de bande passante ne peut être strictement supérieure à 1. Il ne peut y a voir qu'une unique valeur de proportion de bande passante spécifiée pour chaque niveau de priorité. La somme des proportion de bande passante spécifiées pour chaque niveau de priorité ne peut être strictement supérieure à 1. Ces trois règles sont spécifiées par les équations 5.1, 5.2 et 5.3, dans lesquelles α_i^+ représente la proportion de bande passante allouée au niveau de priorité i. Elles sont des conditions nécessaires mais pas suffisantes pour garantir le respect des contraintes temps réel des flux de données gérés par ce mécanisme d'ordonnancement.

$$\forall i \in \mathbb{N} | i \leq 7, \alpha_i^+ \leq 1 \tag{5.1}$$

$$\forall i, j \in \mathbb{N} | i, j \leq 7, i = j \Rightarrow \alpha_i^+ = \alpha_j^+ \tag{5.2}$$

$$\sum_{i=0}^{7} \alpha_i^+ \leq 1 \tag{5.3}$$

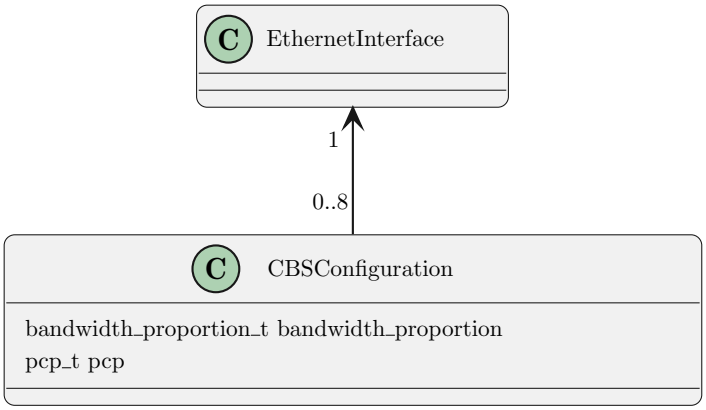


FIGURE 5.8 – Diagramme de classe de la ressource de modélisation servant à modéliser la configuration du CBS.

5.3 MARTE – *Generic Resource Modeling*

Notre approche de modélisation est inspirée de MARTE [MAR19], le profil UML (*Unified Modeling Language*) normalisé par l'OMG (*Object Management Group*) pour la spécification de systèmes temps réel embarqués. Plus particulièrement, nous nous sommes inspirés du chapitre *Generic Resource Modeling*, qui définit deux concepts sur lesquels nous nous basons : les *Computing Resource* et les *Communication Media*. La spécification MARTE étant bien connue dans le domaine des systèmes temps réels embarqués, elle constitue un bon point de départ pour notre approche de modélisation de réseau Ethernet déterministes et embarqués.

Nous n'utilisons donc pas l'entièreté de la spécification MARTE mais nous avons sélectionné les parties qui correspondent à notre besoin. Nous utilisons le concept de *Computing Resource*

pour représenter un nœud du réseau, il permet donc de modéliser un pont TSN ou un terminal. Nous utilisons le concept de *Communication Media* afin de représenter les éléments du réseau qui connectent les nœuds ou qui leur sont liés, il permet donc de modéliser les liens Ethernet et les flux de données.

En matière de terminologie, notre approche de modélisation utilise la notion de « ressource », répartie en deux catégories correspondant aux concepts tirés de MARTE : *ComputationResource*, abrégée en *compRsc*, et *CommunicationResource*, abrégée en *commRsc*.

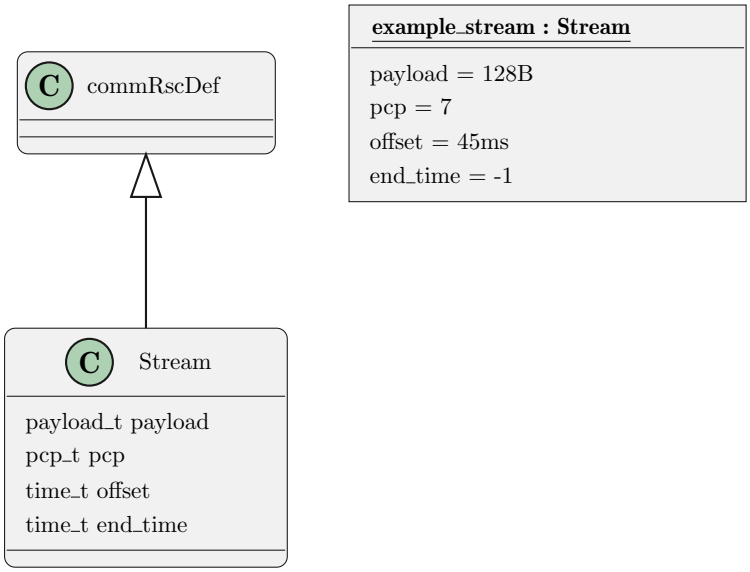


FIGURE 5.9 – Diagramme représentant la définition d'un flux de données, équivalente à une classe, et son instanciation, équivalente à un objet.

En suivant les principes de la programmation orientée objet, nous faisons une différence entre la définition et l'instanciation des modèles des éléments du réseau : les définitions sont appelées *compRscDef* et *commRscDef* et les instanciations sont appelées *compRsc* et *commRsc*. Par exemple, la définition d'un flux de données est un *commRscDef* qui contient tous les différents paramètres nécessaires à sa caractérisation, comme illustré par la figure 5.9. L'instanciation de ce flux de données est un *commRsc* qui utilise cette définition comme référence et spécifie les valeurs de chaque paramètre. Il peut y avoir plusieurs instances faisant référence à la même définition.

Cette séparation entre définition et instanciation permet à notre solution de modélisation de mieux s'intégrer avec Sigil-UCM [Ver23], qui est un générateur de code développé dans les laboratoires de Thales Research & Technology France, qui n'a pas l'utilité des valeurs des paramètres et ne requiert donc que les définitions. Cet outil est présenté dans le chapitre 3 ainsi que dans l'annexe B.

5.4 Ressources de modélisation de réseaux TSN

Pour modéliser les différents éléments du réseau, nous utilisons une syntaxe XML, pour sa flexibilité et parce que la suite des travaux présentés dans cette thèse, dans le chapitre 6, s'appuie en partie sur le standard UCM [UCM21] qui définit une syntaxe XML que nous avons étendue. Les fichiers produits, qui contiennent le modèle du réseau dans ce format, serviront ensuite

d'entrée pour MoBACT, notre outil de génération de configuration.

Nous pouvons distinguer deux catégories d'éléments à modéliser : les éléments de la topologie et les éléments de communication. La topologie est spécifiée par un ensemble d'éléments comprenant : les terminaux, les ponts et les liens. Les communications sont modélisées à l'aide des flux de données et des exigences système.

5.4.1 Modélisation de la topologie

Modélisation des terminaux

```
<commPortDef name="Ethernet_interface">
  <configParam name="bandwidth" type="types::bandwidth_t"/>
</commPortDef>
<commPortDef name="Eth_IP_interface">
  <extends ref="Ethernet_interface"/>
  <configParam name="IP_address" type="types::IP_t"/>
  <configParam name="MAC_address" type="types::MAC_address_t"/>
</commPortDef>
<computRscDef name="End_Point">
  <computRscPort def="Eth_IP_interface" name="ep_eth0"/>
</computRscDef>
```

Listing 5.1 – Modèle de définition des ressources servant à modélisser les terminaux et les interfaces Ethernet.

Le listing 5.1 présente la modélisation de la ressource permettant de modéliser les terminaux dans la syntaxe XML que nous avons définit.

```
<computRsc def="::ethernet_rsc::End_Point" name="controle">
  <computPortConf>
    <computPortRef ref="ep_eth0"/>
    <config def="IP_address" value="{&quot;192.168.0.1&quot;,&quot;/24&
    ↪ quot;}"/>
    <config def="MAC_address" value="{&quot;00:00:00:00:00:01&quot;}"/>
    <config def="bandwidth" value="{&quot;100&quot;,&quot;Mbps&quot;}"/>
  </computPortConf>
</computRsc>
```

Listing 5.2 – Modèle de l'instanciation du terminal sur lequel est déployé le composant *controle*.

Le listing 5.2 présente l'instanciation du terminal *controle*, responsable de transmettre les consignes de limitation de vitesse au moteur comme expliqué dans le chapitre 4. Ce terminal possède une unique interface Ethernet dont la valeur des paramètres est fournie.

Modélisation des ponts

Le listing 5.3 présente la modélisation de la ressource permettant de modéliser les terminaux dans la syntaxe XML que nous avons définit.

```
<computRscDef name="TSN_Switch">
  <computRscPort def="::ethernet_rsc::Ethernet_interface" max="1"
   ↪ min="0" name="ts_eth0"/>
  <computRscPort def="::ethernet_rsc::Ethernet_interface" max="1"
   ↪ min="0" name="ts_eth1"/>
  <computRscPort def="::ethernet_rsc::Ethernet_interface" max="1"
   ↪ min="0" name="ts_eth2"/>
  <configParam max="-1" min="0" name="supported_functionalities"
   ↪ type="types::supported_functionalities"/>
  <configParam name="processing_delay" type="types::time_t"/>
</computRscDef>
<computRscDef name="Ethernet_Switch">
  <computRscPort def="Ethernet_interface" max="2" min="-1"
   ↪ name="es_ethX"/>
</computRscDef>
```

Listing 5.3 – Modèle de définition de la ressource servant à modéliser les ponts TSN.

```
<computRsc def="::tsn_rsc::TSN_Switch_4_Ports" name="pont2">
  <comment> Model: LS1028ARDB</comment>
  <config def="processing_delay" value="{1, us}"/>
  <computPortConf>
    <computPortRef ref="es_eth0"/>
    <config def="bandwidth" value="{100, Mbps}"/>
  </computPortConf>
  <computPortConf>
    <computPortRef ref="es_eth1"/>
    <config def="bandwidth" value="{100, Mbps}"/>
  </computPortConf>
  <computPortConf>
    <computPortRef ref="es_eth2"/>
    <config def="bandwidth" value="{100, Mbps}"/>
  </computPortConf>
</computRsc>
```

Listing 5.4 – Modèle de l'instanciation du pont TSN *pont2*.

Le listing 5.4 présente l'instanciation du pont TSN *pont2* de la figure 5.6. Ce pont comporte trois ports dont la valeur des paramètres de bande passante est de 100 Mb/s, la valeur de son paramètre de latence de commutation est de 1 µs.

Modélisation des liens

```
<commRscDef name="Ethernet_Link">
  <commRscPort def="Ethernet_interface" max="2" min="2" name="
    ↪ ethernet_interfaces_el"/>
  <configParam name="propagation_delay" type="::tsn_rsc::types::time_t"/
    ↪ >
</commRscDef>
```

Listing 5.5 – Modèle de définition de la ressource servant à modéliser les liens Ethernet.

Le listing 5.5 présente la modélisation de la ressource permettant de modéliser les liens Ethernet dans la syntaxe XML que nous avons définie.

```
<commRsc def="::ethernet_rsc::Ethernet_Link" name="controle_to_pont2">
  <config def="propagation_delay" value="{0, ms}"/>
  <commPort computRsc="controle" computRscPort="ep_eth0" name="
    ↪ controle_eth0"/>
  <commPort computRsc="pont2" computRscPort="ts_eth0" name="pont2_eth0"/
    ↪ >
</commRsc>
```

Listing 5.6 – Modèle de l'instanciation du lien Ethernet reliant *controle* à *pont2*.

Le listing 5.6 présente l'instanciation du lien Ethernet *controle_to_pont2*. Ce lien relie deux interfaces Ethernet entre elles et la valeur de son paramètre de délais de propagation est fournie.

5.4.2 Modélisation des communications

Modélisation des flux de données

Le listing 5.7 présente la modélisation de la ressource permettant de modéliser un flux de données dans la syntaxe XML que nous avons définie.

Le listing 5.8 présente l'instanciation du flux de données permettant la transmission de la nouvelle consigne à respecter entre le composant *controle* et le composant *moteur*.

Ce flux de données appartient au niveau de priorité le plus élevé, le niveau 7. Il est transmis de l'interface *eth0* du terminal *Controle* vers l'interface *eth0* du terminal *Moteur* et sa charge utile est de 38 octets. C'est un flux périodique dont la période de transmission est de 30 ms et la phase de transmission est de 11002 µs.

```
<commRscDef name="Applicative_Stream">
  <configParam name="payload" type="types::payload_t"/>
  <configParam name="pcp" type="types::pcp_t"/>
  <configParam min="0" name="offset" type="types::time_t"/>
  <configParam min="0" name="end_time" type="types::time_t"/>
  <rscParam max="1" min="0" name="listener">
    <allowedRscDef ref="::ethernet_rsc::Eth_IP_interface"/>
  </rscParam>
  <rscParam max="1" min="0" name="talker">
    <allowedRscDef ref="::ethernet_rsc::Eth_IP_interface"/>
  </rscParam>
</commRscDef>
<commRscDef name="Periodic_Stream">
  <extends ref="Applicative_Stream"/>
  <configParam name="period" type="types::time_t"/>
</commRscDef>
<commRscDef name="Sporadic_Stream">
  <extends ref="Applicative_Stream"/>
  <configParam name="min_interval" type="types::time_t"/>
</commRscDef>
```

Listing 5.7 – Modèle de définition de la ressource servant à modéliser les flux de données.

```
<commRsc def="::tsn_rsc::Periodic_Stream" name="
    ↪ consigne_conn_controle1_consigne_moteur_emtr_pe_to_moteur1">
  <config def="pcp" value="7"/>
  <config def="payload" value="{38, B}"/>
  <config def="offset" value="{11002, us}"/>
  <config def="period" value="{30000, us}"/>
  <rscConfig def="talker" value="::model::topologie1::controle_to_pont2.
    ↪ controle_eth0"/>
  <rscConfig def="listener" value="::model::topologie1::moteur_to_pont1.
    ↪ moteur_eth0"/>
</commRsc>
```

Listing 5.8 – Modèle de l'instanciation du flux de données permettant la tranmission de la nouvelle consigne de vitesse à respecter entre *controle* et *moteur*.

Modélisation des exigences système

Le listing 5.9 présente la modélisation de la ressource permettant de modéliser les exigences système qui s'appliquent aux flux de données dans la syntaxe XML que nous avons définie.

Le listing 5.10 présente l'instanciation des exigences système qui s'appliquent au flux de données présenté dans le listing 5.8. Ces exigences indiquent que ce flux de données passe par l'interface *eth0* du pont *pont2* puis par l'interface *eth2* du pont *pont1*. Il indique également que l'identifiant du VLAN sur lequel ce flux de données sera transmis est le VLAN 1. Les exigences temps réel que le réseau doit être capable de garantir pour ce flux de données sont fournies : son échéance est fixée à 10 ms et sa gigue à 1 ms. Enfin, il est indiqué que ce flux de données nécessite le placement d'une *guard band* avant la fenêtre temporelle du TAS pendant lequel il sera transmis.

```
<commRscDef name="Stream_System_Requirements">
  <rscParam max="-1" min="1" name="Stream">
    <allowedRscDef ref="Applicative_Stream"/>
  </rscParam>
  <configParam min="0" name="deadline" type="types::time_t"/>
  <configParam name="vlan_id" type="types::vlan_id_t"/>
  <configParam min="0" name="maximum_jitter" type="types::time_t"/>
  <configParam defaultValue="false" name="is_critical"
   ↪ type="types::is_critical_t"/>
  <rscParam max="1" min="0" name="overridden_listener">
    <allowedRscDef ref="::ethernet_rsc::Eth_IP_interface"/>
  </rscParam>
  <rscParam max="1" min="0" name="overridden_talker">
    <allowedRscDef ref="::ethernet_rsc::Eth_IP_interface"/>
  </rscParam>
  <structParam max="-1" min="1" name="path">
    <rscParam max="-1" min="0" name="path_member">
      <allowedRscDef ref="::ethernet_rsc::Ethernet_interface"/>
    </rscParam>
  </structParam>
</commRscDef>
```

Listing 5.9 – Modèle de définition de la ressource servant à modéliser les exigences systèmes qui s'appliquent aux flux de données.

```
<commRsc def="::tsn_rsc::Stream_System_Requirements" name="stream2">
  <rscConfig def="Stream" value="::streams::
   ↪ consigne_conn_controle1_consigne_moteur_emtr_pe_to_moteur1"/>
  <rscConfig def="overridden_talker" value="::model::topologie1::
   ↪ controle_to_pont2.controle_eth0"/>
  <rscConfig def="overridden_listener" value="::model::topologie1::
   ↪ moteur_to_pont1.moteur_eth0"/>
  <structConfig def="path">
    <rscConfig def="path_member" value="::model::topologie1::
   ↪ controle_to_pont2.pont2_eth0"/>
    <rscConfig def="path_member" value="::model::topologie1::
   ↪ pont1_to_pont2.pont1_eth2"/>
  </structConfig>
  <config def="vlan_id" value="1"/>
  <config def="deadline" value="{10,ms}"/>
  <config def="maximum_jitter" value="{1,ms}"/>
  <config def="is_critical" value="true"/>
</commRsc>
```

Listing 5.10 – Modèle de l'instanciation des exigences systèmes qui s'appliquent à un flux de données.

```
<struct name="time_slot_t">
  <field name="duration" type="time_t"/>
  <field name="allowed_priorities" type="pcp_seq_t"/>
</struct>
```

Listing 5.11 – Modèle de définition de la ressource servant à modéliser une fenêtre temporelle de la configuration du TAS d'un port.

5.4.3 Modélisation de la configuration des mécanismes d'ordonnancement

Modélisation de la configuration du TAS

Le listing 5.11 présente la modélisation de la ressource permettant de modéliser la configuration du TAS portée par les interfaces Ethernet dans la syntaxe XML que nous avons définit.

```
<computRsc def="::tsn_rsc::TSN_Switch_4_Ports" name="pont2">
  [...]
  <computPortConf>
   <computPortRef ref="ts_eth2"/>
   <config def="bandwidth" value="{100.0, Mbps}"/>
   <structConfig def="TAS_configuration">
    <config def="cycle_duration" value="{30000, us}"/>
    <config def="time_slots" value="{{11007, us}, [6, 5, 4, 3, 2, 1, 0]}
    ↪ "/>
    <config def="time_slots" value="{{11, us}, []}"/>
    <config def="time_slots" value="{{10, us}, [7]}"/>
    <config def="time_slots" value="{{18972, us}, [6, 5, 4, 3, 2, 1, 0]}
    ↪ "/>
   </structConfig>
  </computPortConf>
  [...]
</computRsc>
```

Listing 5.12 – Modèle de l'instanciation d'une configuration du TAS sur le port *eth2* de *pont2*.

Le listing 5.12 présente l'instanciation d'une configuration du TAS sur un port d'un pont. Les différentes fenêtres temporelles formant le cycle TAS sont définies et leur durée ainsi que les niveaux de priorité des trames ayant la possibilité d'être transmises pour chacune d'entre elles.

La figure 5.10 présente la configuration du TAS obtenue pour le port *eth2* du pont *pont2*, ce port relie *pont2* à *pont1*.

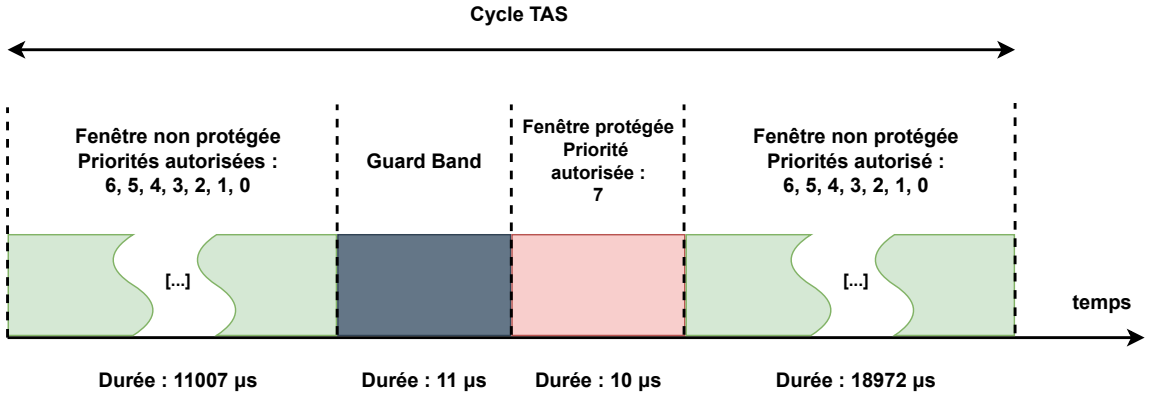


FIGURE 5.10 – Diagramme des fenêtres temporelles du cycle TAS présentés dans le listing 5.12.

Modélisation de la configuration du CBS

```
1 <struct name="idle_slope_t">
2   <field name="bandwidth_proportion" type="bandwidth_proportion_t"/>
3   <field name="pcp" type="pcp_t"/>
4 </struct>
```

Listing 5.13 – Modèle de définition de la ressource servant à modéliser la configuration du CBS pour un niveau de priorité.

Le listing 5.13 présente la modélisation de la ressource permettant de modéliser la configuration du CBS portée par les interfaces Ethernet dans la syntaxe que nous avons définit.

```
1 <computRsc def="::test_def::Ethernet_Switch_4_Ports" name="pont2">
2   [...]
3   <computPortConf>
4     <computPortRef ref="es_eth0"/>
5     <config def="bandwidth" value="{100, Mbps}"/>
6     <config def="CBS_configuration" value="{0.25, pcp_0}"/>
7   </computPortConf>
8   [...]
9 </computRsc>
```

Listing 5.14 – Modèle de l'instanciation d'une configuration CBS sur le port *eth0* de *pont2*.

Le listing 5.14 présente l'instanciation de la configuration du CBS sur un port d'un pont. La proportion de bande passante et le niveau de priorité auquel elle est allouée est spécifiée.

5.5 Discussion et conclusion

Dans ce chapitre, nous avons présenté notre approche de modélisation des éléments d'un réseau TSN. Cette approche propose un ensemble de ressources de modélisation qui permettent de représenter chacun des éléments d'un réseau TSN : les ponts, les terminaux, les liens et les

flux de données. Ces ressources doivent être instanciées pour créer des modèles de réseau TSN dans le format utilisé par notre outil de génération de configuration, MoBACT, présenté dans le chapitre 8.

Comme on peut le constater dans la définition des ressources, l'élément le plus complexe est la définition des flux de données. C'est l'élément central du modèle car il contient les informations les plus importantes pour la génération de la configuration, le reste du modèle ne donnant que la topologie et la bande passante disponible. La génération automatique du modèle des flux de données semble donc être une fonctionnalité très importante qui permettrait également d'éviter les erreurs.

De même que pour les ressources de modélisation, leur instanciation se fait dans la syntaxe XML que nous avons définie. Le choix d'utiliser XML plutôt que d'autres format permettant le stockage de données a été motivé par les raisons suivantes : la facilité pour étendre la syntaxe, la relative clarté des fichiers produits (par rapport à un format comme JSON par exemple) et par des raisons de compatibilité avec des outils déjà existants, notamment Sigil-UCM [Ver23].

Il est donc possible d'écrire des modèles de réseaux TSN à la main en utilisant la syntaxe que nous proposons mais les ressources ont été crées de façon à être facile à utiliser avec *Eclipse Modeling Framework* (EMF) [EMF]. EMF est une armature logicielle de modélisation qui permet de visualiser et d'éditer des modèles ainsi que de les manipuler en générant un ensemble de classes Java représentant le modèle.

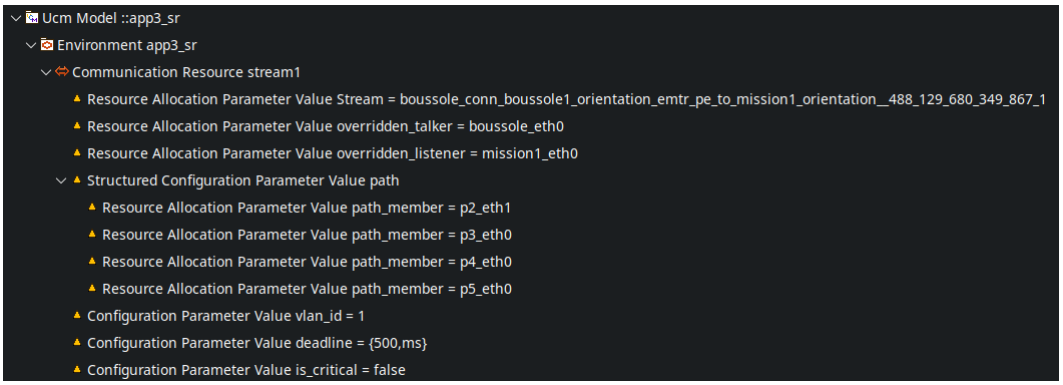


FIGURE 5.11 – Exemple d'utilisation de l'éditeur de modèle arborescent d'Eclipse EMF.

EMF met à disposition un éditeur de modèle arborescent comme visible sur la figure 5.11. C'est l'utilisation de cet éditeur qui a mené à la décision de créer les ressources permettant de représenter les ponts et les terminaux avec un nombre de ports fixe au lieu d'un nombre de ports arbitraires. Ce nombre de ports fixe permet de facilement faire référence aux différents ports dans les autres éléments du modèle du réseau. Il existe donc plusieurs ressources permettant la modélisation des ponts et des terminaux ayant chacune un nombre de ports différents et il est très facile d'en ajouter pour avoir le nombre de ports voulu.

Une autre raison ayant menée à l'utilisation d'EMF est la possibilité de mettre en place un ensemble de règles de validation. Ces règles permettent de vérifier qu'un modèle de réseau TSN créé respecte bien les contraintes imposées par chaque ressource de modélisation. Par exemple, une de ces règles vérifie qu'un lien Ethernet ne possède bien que deux extrémités et que ses extrémités sont des ports de pont ou de terminaux qui existent dans le modèle.

Ces règles permettent donc de vérifier le respect des cardinalités et des types spécifiés pour chaque relation entre les ressources de modélisation. Elles peuvent également être utilisés pour vérifier le respect d'autres contraintes comme les équations 5.1, 5.2 et 5.3.

La limitation à un unique récepteur pour chaque flux de données a une double origine : la limitation de la complexité du problème et le fait que la bibliothèque de communication utilisée dans le cadre de nos expérimentations ne supporte que des *socket unicast*. Néanmoins, le modèle de ressources a initialement été créé avec le support des communications *multicast* en offrant la possibilité de créer des modèles dans lesquels les flux de données pouvaient avoir plusieurs destinataires et pouvaient emprunter des chemins différents. Les contraintes industrielles qui ont conduit à cette restriction n'empêchent donc pas définitivement le support des communications *multicast* par notre modèle de ressources et la restriction peut facilement être levée.

Le système étudié n'a donc pas l'usage de flux possédant de multiple destinataires et supporter cette possibilité serait la source d'une complication de modélisation puisqu'il faudrait que l'utilisateur crée plusieurs couples chemin destinataire pour chaque flux de données. Nous considérons que cette possibilité supplémentaires ne vaut pas l'augmentation de la difficulté d'utilisation de notre approche et que son absence ne remet pas en cause son intérêt.

La séparation des caractéristiques des flux de données en deux parties – la partie contenant les caractéristiques intrinsèques des flux de données et la partie contenant les exigences du système – permet de définir les paramètres qui devront être calculés automatiquement et ceux qui ne tiennent qu'aux exigences du système, c'est-à-dire les chemins et les contraintes de latence et de gigue par exemple.

Le calcul automatique de ces paramètres sera présenté dans le chapitre 6.

Chapitre 6

Déduction automatique du modèle des flux de données

Comme expliqué dans le chapitre 5, la production automatique du modèle des flux de données est une fonctionnalité très importante dans le processus de conception et de configuration d'un réseau TSN. En effet, cette étape est à la fois difficile et centrale pour le niveau de confiance qu'il est possible d'accorder à une configuration d'un réseau TSN.

C'est une étape difficile car l'obtention des valeurs des paramètres utilisés pour représenter les flux de données dans une approche de modélisation telle que celle que nous proposons dans cette thèse dépend de l'application qui utilisera le réseau. Certaines de ces valeurs, comme la phase de transmission des flux de données, dépendent du flot d'exécution de l'application, ce qui peut être difficile à suivre sans un processus automatisé.

De plus, il est vital pour que la configuration du réseau produite ait un quelconque intérêt que la cohérence entre les flux de données réels, produits par l'application, et les flux de données présents dans le modèle utilisé pour produire la configuration soit garantie. La moindre erreur de modélisation qui briserait cette cohérence rendrait la configuration produite inutile car elle ne correspondrait pas à la réalité des flux de données.

Dans ce chapitre, nous présentons notre méthode de modélisation automatique des flux de données. Afin de garantir la cohérence entre les flux de données réels de l'application et ceux présents dans le modèle, nous présentons notre utilisation d'UCM et de ses extensions présentes dans [Ver23]. Enfin, nous expliquons comment les valeurs des paramètres servant à la modélisation des flux de données sont calculées. Les deux contributions présentées dans ce chapitre sont donc : le métamodèle contenant les informations dont nous avons besoin et la méthode utilisée pour générer automatiquement le modèle des flux de données.

6.1 Notre approche de modélisation automatique du modèle des flux de données

Les outils de simulation et de génération de configuration supposent que les flux de données sont connus, qu'ils font partie des données du problème. Par exemple, NeSTiNg et RTaW-Pegase prennent les informations concernant les flux de données comme entrée, comme présenté dans la section 3.3. Or, même si l'utilisation de ces outils permettait de produire la meilleure configuration du réseau possible pour ces données d'entrée, si elles ne correspondent pas à la réalité des flux de données réellement produits par l'application qui utilise le réseau, ce serait parfaitement inutile.

Notre pratique de l'utilisation de ces outils servant à la conception et à la vérification de systèmes, dans un cadre industriel, a démontré que l'obtention de ces informations n'est pas une étape simple de la conception d'un système, notamment dans le cas du contrôle-commande.

Un flux multimédia, comme un flux vidéo, est un flux dont les caractéristiques sont généralement connues car elles peuvent se résumer à un débit. De plus, ce type de flux ne fait pas typiquement partie des flux de données critique. En revanche, un flux de contrôle-commande n'est pas aussi régulier, pas aussi volumineux, d'un niveau de criticité supérieur et peut dépendre de l'exécution d'autres parties du système. De part leur criticité, des exigences temporelles sont associées aux flux de contrôle-commande : leur latence et leur gigue, contrairement aux flux multimédia. Dans le cadre d'un réseau temps réel, il est donc absolument nécessaire de pouvoir déterminer avec précision ses caractéristiques temporelles : période et phase de transmission, tout comme ses caractéristiques de volume : la taille des données transmises à chaque émission d'un flux de données dans notre cas.

La façon que nous avons choisi pour résoudre ce problème est de supprimer à l'utilisateur la responsabilité de caractériser les flux de données en outillant ce processus.

Pour cela, nous avons lié deux approches de modélisation : celle des applications et celle du réseau. Nous avons donc lié l'approche de modélisation logicielle présentée dans le chapitre 3, qui utilise UCM, et notre approche de modélisation réseau présentée dans le chapitre 5. Cette liaison des deux approches permet de passer d'un modèle de l'architecture logicielle à un modèle des flux de données tels que spécifiés dans le chapitre 5.

Cela permet, lors de changement dans l'architecture du système, de n'avoir à effectuer les modifications que dans l'unique modèle central des applications. Ces modifications seront automatiquement répercutées dans la modélisation des flux de données et dans le code généré des applications, comme nous l'illustrerons dans le chapitre 9.

De plus, UCM permet d'obtenir du code exécutable à partir d'un modèle d'architecture, pour cela le standard définit un modèle intermédiaire qui correspond à la structure du code de l'application [UCM21]. Nous appliquons la même démarche pour le calcul du modèle des flux de données à partir de l'architecture logicielle.

Cette combinaison des approches de modélisation et de génération automatique nous permet de garantir la cohérence entre les flux de données produits par l'application et ceux présents dans le modèle utilisé pour générer la configuration du réseau qui sera utilisé par l'application, comme présenté dans la figure 6.1. La production de l'armature logicielle contrôlant l'exécution des composants (e.g. leur période et leur phase d'exécution) et du modèle des flux de données venant du même modèle central permet d'apporter cette garantie.

Ces caractéristiques temporelles peuvent être difficiles à obtenir sans faire d'erreurs dans des cas où la production d'un flux de données par un composant dépend de son exécution qui elle-même dépend de la réception d'un flux de données transmis suite à l'exécution d'un autre composant. Il est très facile de faire des erreurs dans ce genre de cas, tout comme il est facile de faire des erreurs dans le calcul de la phase de transmission d'un flux de données lorsqu'elle dépend d'une spécification de comportement qui contient des boucles et des alternatives.

De la même façon que les caractéristiques temporelles des flux de données peuvent être déduite du modèle, la taille des données peut également être calculée automatiquement. Il est possible d'estimer grossièrement la taille des données mais pour avoir une information fiable il faut pouvoir automatiser ce processus.

Le modèle intermédiaire présent dans la figure 6.1 est un modèle de la sémantique d'exécution et de communication et il sera présenté dans la section 6.2. Le processus de calcul de la valeur des paramètres des flux de données sera présenté dans la section 6.3.

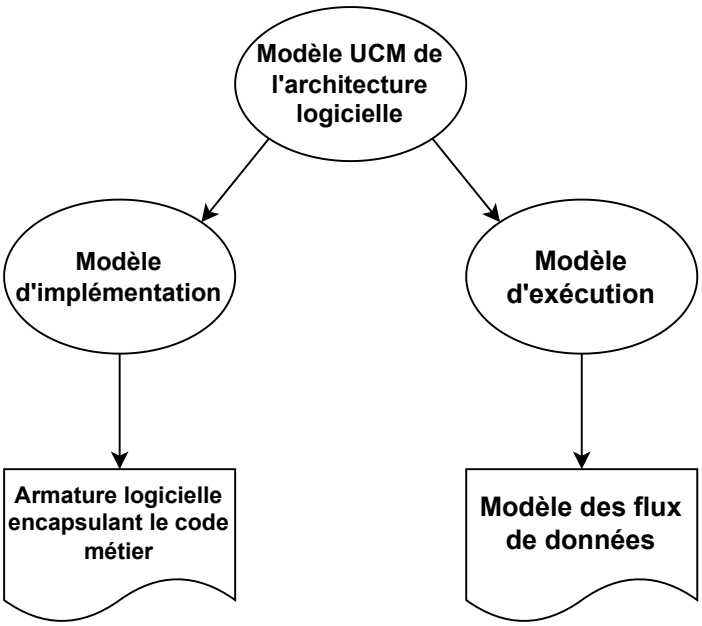


FIGURE 6.1 – Schéma présentant les différentes productions obtenues automatiquement à partir de la modélisation de l'architecture logicielle.

6.2 Sémantique d'exécution et de communication

La production automatique du modèle des flux de données nécessite une spécification préalable du comportement des applications et que cette spécification soit exploitable pour atteindre cet objectif. Nous lions donc notre approche de modélisation du réseau, présentée dans le chapitre 5, à une approche de modélisation des applications qui utiliseront le réseau reposant sur le standard UCM présenté dans la section 3.2. Cette approche et son outillage nous permet de rassembler les informations dont nous avons besoin pour calculer automatiquement un modèle des flux de données.

Ce processus se déroule en plusieurs étapes : la spécification du comportement des applications, l'assemblage de cette spécification avec les éléments constituant la plateforme d'exécution et enfin le calcul du modèle des flux de données. L'étape d'assemblage de la spécification utilise un métamodèle intermédiaire et cette section a pour but de le présenter et d'expliquer la traduction des concepts utilisés en UCM vers ce métamodèle.

6.2.1 Spécification du comportement des applications

Dans la suite, nous ferons référence à « la spécification du comportement des composants », ceci désigne la spécification des éléments de code métier des composants, c'est-à-dire les méthodes qui contiennent la logique de l'application, par opposition aux éléments de plateforme – politiques techniques et connecteurs – qui sont eux implémentés dans une bibliothèque dédiée.

Le standard UCM n'intègre pas la spécification du comportement des composants qui forment les applications. On peut se représenter un composant UCM comme une boîte à laquelle on peut associer du code, mais les éléments concernant son exécution et ses possibilités de communication sont extérieurs au composant.

L'outil Sigil-UCM étend le standard UCM en offrant à l'utilisateur la possibilité de spécifier le comportement des composants. Pour ce faire, il propose un ensemble de constructions inspirées

d'un sous ensemble des diagrammes de séquences définis par UML [UML17] : les étapes de calcul et de communication, les séquences, les alternatives et les boucles.

La spécification du comportement des composants prend la forme d'une séquence d'étapes de calcul et d'étapes d'appel de méthode. Un meilleur temps d'exécution ($BCET$) et un pire temps d'exécution ($WCET$) sont associés à chacune de ces étapes. Ces séquences d'étapes permettent donc de spécifier le comportement de chaque méthode d'un composant.

Une étape de calcul représente un traitement réalisé sur le terminal sur lequel est déployé le composant. Une étape d'appel de méthode peut faire appel aux connecteurs (voir la section 3.2) et peut donc engendrer des communications réseau.

Le sous ensemble des diagrammes de séquence utilisé permet également à l'utilisateur d'employer des constructions comme l'alternative ou la boucle.

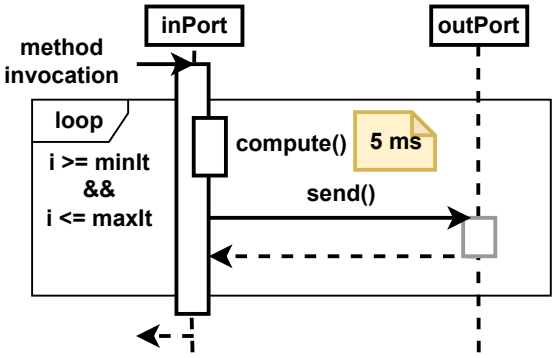


FIGURE 6.2 – Diagramme de séquence d'une boucle comportant des communications réseau.

La figure 6.2 présente l'utilisation des boucles pour la spécification du comportement des composants, le diagramme présente la spécification complète d'une méthode d'un composant. Cette spécification utilise une boucle, *loop*, dont le nombre minimum d'itération est *minIt* et le nombre maximum d'itération est *maxIt*[16]. Cette boucle contient une séquence composée d'une étape de calcul, *compute()*, dont le pire temps d'exécution est 5 ms, suivie d'une étape d'appel de méthode qui appelle la méthode *send()*. Cette étape d'appel de méthode va ensuite passer par un connecteur et engendrer une communication réseau.

Le listing 6.3 présente l'utilisation des alternatives pour la spécification des composants, le diagramme présente la spécification complète d'une méthode d'un composant. Cette spécification utilise une alternative, dont la sémantique est que le flux d'exécution ne peut suivre qu'une seule branche parmi toutes celles disponibles. Cette alternative comporte deux branches et chacune contient une étape d'appel de méthode, *sendA()* pour la première branche et *sendB()* pour la seconde. Ces deux méthodes sont différentes mais engendre toutes les deux une communication réseau en passant par un connecteur.

Le modèle des applications obtenu par la combinaison d'un modèle UCM et d'une spécification du comportement de ses composants permet de décrire la sémantique d'exécution de la partie métier. Cette sémantique doit être combinée avec les sémantiques d'exécution et de communication des connecteurs et des politiques techniques afin de construire la sémantique globale de l'application entière.

16. Il est possible de spécifier un nombre infini d'itérations. En revanche, l'exécution des composants étant gérée par les politiques techniques (comme présenté dans le chapitre 3), une spécification du comportement d'un composant comportant une boucle infinie produisant des communications réseaux aurait pour effet de produire un nombre infini de flux de données à chaque exécution, ce qui n'a pas de sens.

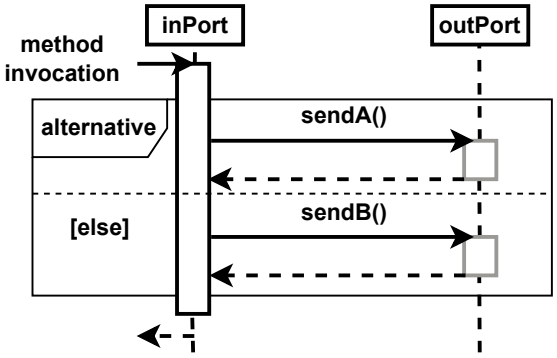


FIGURE 6.3 – Diagramme de séquence d'une alternative comportant des communications réseau.

```
1  <detailed lang="::ucm_lang::cpp::CPP11_typed" name="moteur1" type="::
       ↪ model::comps::moteur">
2    <policyRef ref="exec_psv"/>
3    <compPortImpl name="consigne_rcvr_pe" port="consigne" portElement="
       ↪ rcvr_pe"/>
4    <compPortImpl name="etat_emtr_pe" port="etat" portElement="emtr_pe"/>
5    <methScen meth="push" name="push" port="consigne_rcvr_pe">
6      <seq name="seq">
7        <comput name="calcul">
8          <annotation def="::sigil_ann::tap::ta_step_timing">
9            <config def="bcet" value="{1,ms}"/>
10           <config def="wcet" value="{1,ms}"/>
11         </annotation>
12       </comput>
13       <call meth="push" name="push_etat" port="etat_emtr_pe"/>
14     </seq>
15   </methScen>
16 </detailed>
```

Listing 6.1 – Implémentation détaillée du composant *moteur1* du type *Moteur*.

Pour illustrer ceci, les listings 6.1 et 6.2 présentent des implémentations détaillées des types de composant *Moteur* et *Controle* présentés dans la section 3.2. L'implémentation détaillée du composant *moteur1* comporte la spécification du comportement du code métier, entre la ligne 5 et la ligne 15. Cette spécification concerne une méthode appelée *push*, qui contient une séquence d'une étape de calcul, ligne 7, suivie d'une étape d'appel de méthode, ligne 13. L'étape de calcul possède une annotation indiquant sa durée d'exécution, qui est ici d'1 ms. De la même façon, entre la ligne 10 et la ligne 28, l'implémentation détaillée du composant *controle1* spécifie le comportement de sa méthode *run*, qui contient une séquence d'une étape de calcul, d'un appel de méthode, d'une autre étape de calcul et enfin d'un dernier appel de méthode.

Il apparaît donc que la spécification du comportement de ce composant se limite à la spécification de ses méthodes. Elle ne comporte pas d'informations à propos de son exécution, qui sont elles portées par la politique technique qui lui est associée. Elle ne comporte pas non plus d'informations concernant les conséquences des appels aux méthodes *push* et *run*, qui sont portées par le connecteur.

```
<detailed lang="::ucm_lang::cpp::CPP11_typed" name="controle1" type="::
    ↪ model::comps::controle">
  <policyRef ref="exec_psv"/>
  <policyRef ref="exec_periodique"/>
  <compPortImpl name="consigne_mission_rdr_pe" port="consigne_mission"
    ↪ portElement="rdr_pe"/>
  <compPortImpl name="consigne_mission_notif_pe" port="consigne_mission"
    ↪  portElement="notif_pe"/>
  <compPortImpl name="consigne_moteur_emtr_pe" port="consigne_moteur"
    ↪ portElement="emtr_pe"/>
  <compPortImpl name="etat_moteur_rdr_pe" port="etat_moteur" portElement
    ↪ ="rdr_pe"/>
  <compPortImpl name="etat_moteur_notif_pe" port="etat_moteur"
    ↪ portElement="notif_pe"/>
  <techPortImpl name="exec_periodique_activation" policy="
    ↪ exec_periodique" portElement="activation"/>
  <methScen meth="run" name="run" port="exec_periodique_activation">
    <seq name="seq">
      <call meth="read_data" name="read_mission" port="
    ↪ consigne_mission_rdr_pe"/>
      <comput name="calcul">
        <annotation def="::sigil_ann::tap::ta_step_timing">
          <config def="bcet" value="{1,ms}"/>
          <config def="wcet" value="{1,ms}"/>
        </annotation>
      </comput>
      <call meth="read_data" name="read_etat" port="etat_moteur_rdr_pe"/
    ↪ >
      <comput name="calcul2">
        <annotation def="::sigil_ann::tap::ta_step_timing">
          <config def="bcet" value="{8,ms}"/>
          <config def="wcet" value="{10,ms}"/>
        </annotation>
      </comput>
      <call meth="push" name="push_consigne" port="
    ↪ consigne_moteur_emtr_pe"/>
    </seq>
  </methScen>
</detailed>
```

Listing 6.2 – Implémentation détaillée du composant *controle1* du type *Controle*.

Il est donc nécessaire de passer par une étape de traduction du modèle UCM et de la spécification du comportement des applications vers un modèle qui permet de représenter, en plus du comportement des composants, la sémantique d'exécution et de communication des composants d'une façon bien adaptée à la réalisation du calcul du modèle des flux.

Ce métamodèle, que nous avons appelé *verif_exec*, dont la réalisation a débuté dans le cadre des travaux présentés dans [SV19] et dont le développement a été poursuivi pour cette thèse, permet d'atteindre cet objectif. La représentation de l'exécution et des communications dans le métamodèle *verif_exec* est faite par l'assemblage des constructions produites par la traduction des éléments d'un modèle UCM : les politiques techniques, les composants et leur comportement

et les connecteurs.

6.2.2 Métamodèle *verif_exec*

Le métamodèle *verif_exec* est un modèle d'exécution qui a notamment pour but de permettre la génération de modèles d'analyse de l'application. Il est possible de générer des réseaux de Petri permettant de s'assurer que l'architecture logicielle ne contient pas d'interblocage ou d'autre types de modèles d'analyse, comme présenté dans [SV19]. C'est donc un modèle d'exécution qui se veut plus général que les réseaux de Petri, en contenant par exemple d'avantage d'informations au niveau des nœuds permettant de faire la différence entre des étapes de calcul et des étapes de communication.

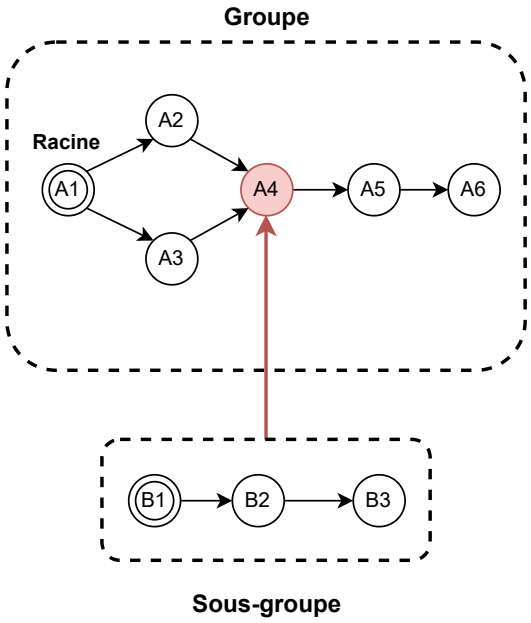


FIGURE 6.4 – Schéma présentant l'organisation des étapes de calcul dans le métamodèle représentant l'exécution des applications.

La figure 6.4 présente la façon dont les étapes de calcul représentant l'exécution des applications sont organisés. Ces étapes sont regroupées en groupes, dont l'étape racine, à partir de laquelle l'exécution commence, est représentée par un double cercle. Les autres cercles représentent le reste des étapes de calcul.

Les étapes

Les étapes peuvent contenir des sous groupes, dont l'organisation est la même, qui représentent un ensemble de sous étapes dont l'exécution doit, généralement, être terminée avant que l'enchaînement des étapes du groupe parent puisse reprendre.

La figure 6.5 présente la notion d'étape qui permet de représenter les différentes étapes de la spécification du comportement des applications dans le métamodèle *verif_exec*.

Les EmStep possèdent les paramètres suivants :

— *bestCET* : entier. Le meilleur temps d'exécution de l'étape.

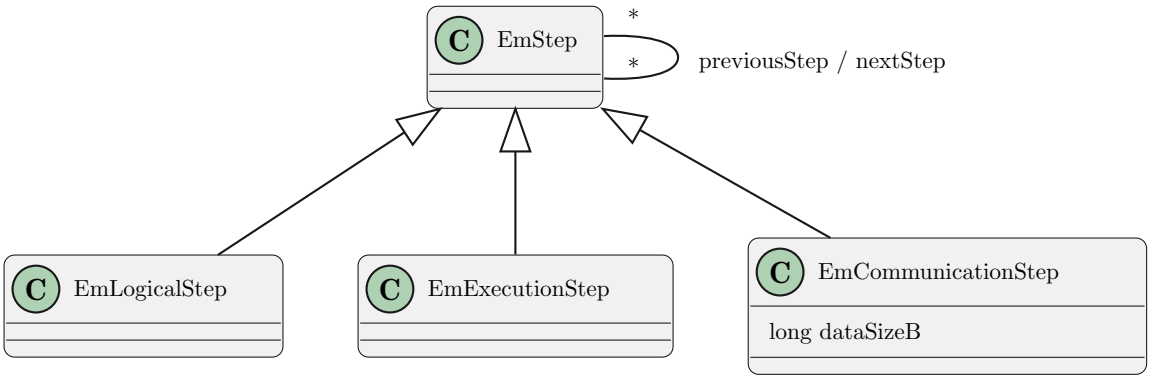


FIGURE 6.5 – Diagramme de classe présentant le concept d'étape dans le métamodèle *verif_exec*.

— *worstCET* : entier. Le pire temps d'exécution de l'étape.

— *nextStep[*]* : EmStep[*]. La liste des EmStep successeurs d'une étape.

— *previousStep[*]* : EmStep[*]. La liste des EmStep prédécesseurs d'une étape.

— *subGroup[0..1]* : EmStepGroup. L'éventuel sous groupe d'une étape.

— *minSubGroupIteration* : entier. Le nombre minimum d'itérations à effectuer si le sous groupe représente le corps d'une boucle.

— *maxSubGroupIteration* : entier. Le nombre maximum d'itérations à effectuer si le sous groupe représente le corps d'une boucle.

— *waitForSubGroup* : booléen. Indique si l'étape doit attendre la fin de l'exécution de son sous groupe avant de reprendre l'exécution du scénario en cours (c'est l'équivalent de la notion de fonction bloquante et non-bloquante).

— *cause* : EmExecutionPattern. Le modèle d'exécution qui s'applique à l'étape et déclenche son exécution.

— *nextStepSynchronization* : EmStepSynchro. Paramètre indiquant la relation de synchronicité d'une étape avec ses successeurs.

— *previousStepSynchronization* : EmStepSynchro. Paramètre indiquant la relation de synchronicité d'une étape avec ses prédécesseurs.

Chaque EmStep indique sa relation de synchronicité avec ses prédécesseurs et ses successeurs. Deux valeurs sont possibles pour ce paramètre : exactement une étape et toutes les étapes. La sémantique de ce paramètre dans le cas des prédécesseurs est la suivante :

— *Exactement une étape* : Si une étape qui précède l'étape courante a terminé son exécution, l'étape courante débute son exécution. Si plusieurs étapes qui précèdent l'étape courant ont terminé leur exécution, l'étape courante débute son exécution autant de fois.

— *Toutes les étapes* : Toutes les étapes qui précèdent l'étape courante doivent avoir terminé leur exécution avant que l'étape courante puisse débuter la sienne.

La sémantique de ce paramètre dans le cas des successeurs est la suivante :

— *Exactement une étape* : Seul un successeur peut être activé, cela mène à la construction d'alternatives ou d'exécutions parallèles.

— *Toutes les étapes* : Toutes les étapes qui succèdent à l'étape courante sont activées lorsqu'elle termine son exécution.

Ce paramètre a un impact important sur la sémantique d'exécution d'un scénario. Reprenons la figure 6.4, suivant la valeur de ce paramètre, l'étape A1 peut mener à des exécutions différentes, listées dans la table 6.1 :

Synchronisation	Exécutions possibles	Explications
Exactement une étape	Soit A1, A2, ... Soit A1, A3, ...	A1 puis A2 (pas A3) A1 puis A3 (pas A2)
Toutes les étapes	A1, (A2 | A3), ...	A1 puis A2 et A3 en parallèle

TABLE 6.1 – Table des exécutions possibles en fonction de la valeur du paramètre de synchronisation des successeurs de l'étape A1.

Il est important de noter qu'un ensemble d'étapes menant à une exécution en parallèle de plusieurs étapes ne peut pas être créé à partir de la spécification du comportement d'un composant. En effet, les constructions qu'il est possible d'utiliser pour spécifier le comportement d'un composant – les étapes, les séquences, les alternatives et les boucles – ne permettent pas de parallélisme à l'intérieur d'un composant. Une construction dans le métamodèle *verif_exec* menant à une exécution parallèle est possible et pourra servir à représenter d'autres entités.

De la même façon, suivant la valeur de ce paramètre pour l'étape A4, cela peut mener à plusieurs exécutions, listées dans la table 6.2 :

Synchronisation	Exécutions possibles	Explications
Exactement une étape	A2, A4, ... A3, A4, ... (A2 | A3), (A4 | A4), ...	A2 puis A4 A3 puis A4 A4 s'exécute deux fois
Toutes les étapes	A2, ∅ A3, ∅ (A2 | A3), A4, ...	A2 puis rien, A4 attend A3 A3 puis rien, A4 attend A2 A2 et A3 puis A4

TABLE 6.2 – Table des exécutions possibles en fonction de la valeur du paramètre de synchronisation des prédécesseurs de l'étape A4.

Dans la pratique, la grande majorité des étapes n'ont qu'un unique prédécesseur et un unique successeur.

Les scénarios

La figure 6.6 présente le concept de scénario du métamodèle *verif_exec*. Les scénarios sont la construction de base servant à représenter l'ensemble du comportement des applications. Ils héritent des *EmStepGroup* et contiennent un ensemble d'étapes (*EmStep*). Les étapes peuvent elles-mêmes contenir un sous groupe.

Les *EmStepGroup* possèdent les paramètres suivants :

— *rootStep* : *EmStep*. L'étape racine du groupe, à partir de laquelle l'exécution du groupe débute.

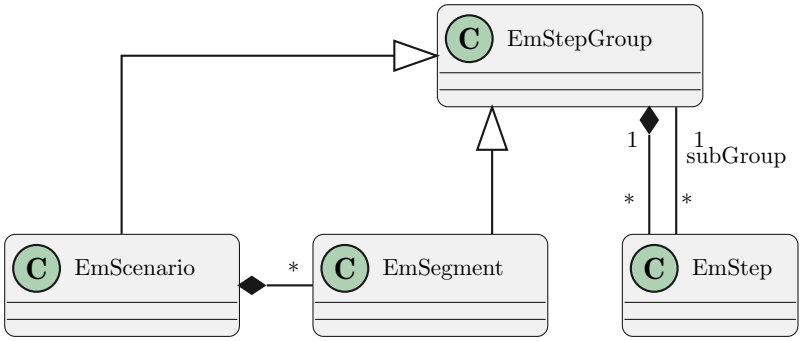


FIGURE 6.6 – Diagramme de classe présentant le concept de scénario dans le métamodèle *verif_exec*.

— *parentStep* : *EmStep*. Si le groupe courant est un sous groupe, ce paramètre référence l'étape qui peut appeler ce groupe. C'est le paramètre opposé du paramètre *subGroup* des *EmStep*.

— *steps[*]* : *EmStep*. La liste des étape qui composent le groupe.

Un *EmSegment* est un groupe, et hérite donc des paramètres des *EmStepGroup*, qui est utilisé pour représenter le corps des boucles qu'il est possible de spécifier dans le comportement des composants. Un segment peut être vu comme un sous scénario spécifique à la représentation des boucles, puisqu'il ne peut exister sans un parent.

Les modèles d'exécution

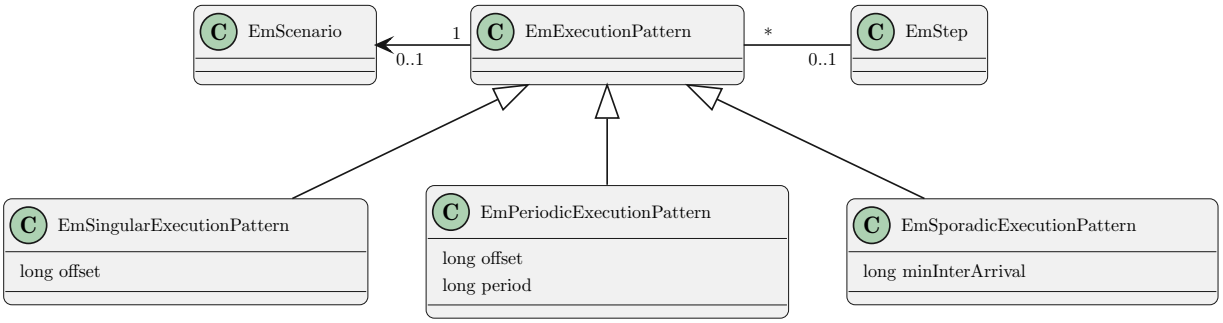


FIGURE 6.7 – Diagramme de classe présentant le modèle d'exécution qui s'applique aux étapes représentant le modèle du comportement des applications dans le métamodèle *verif_exec*.

La figure 6.7 présente les différents modèles d'exécution possibles des scénarios qui sont la cause du déclenchement des étapes qui composent les scénarios.

Les *EmExecutionPattern* possèdent un paramètre *effect*, de type *EmStep*, qui référence l'étape à laquelle ils s'appliquent. Ce paramètre est l'opposé du paramètre *cause* des *EmStep*.

Les modèles d'exécution de type périodique possèdent également des paramètres de période et de phase. Les modèles d'exécution de type unique ne possèdent qu'un paramètre de phase. Enfin, les modèle d'exécution de type sporadique possèdent un paramètre *minInterArrival* qui indique l'intervalle de temps minimum entre deux déclenchements successifs.

6.2.3 Traduction de la spécification des applications vers *verif_exec*

	Point de départ	Application	Communication
UCM	Politiques techniques	Composants Spécification du comportement	Connecteurs
verif_exec	EmScenario EmExecutionStep EmExecutionPattern Appels aux scénarios des composants	EmScenario EmExecutionStep Appels aux scénarios des politiques techniques et des connecteurs	EmScenario EmExecutionStep EmCommunicationStep Appels aux scénarios des composants

TABLE 6.3 – Table présentant la traduction des concepts de modélisation des applications en UCM vers les concepts du métamodèle *verif_exec*.

La table 6.3 présente la traduction du modèle du comportement des applications tels que créé en UCM vers la représentation des concepts représentant le comportement des applications dans le métamodèle *verif_exec*, exploitable pour le calcul automatique du modèle des flux de données.

L'explication de la traduction des concepts utilisés dans un modèle UCM vers ceux utilisés dans le métamodèle *verif_exec* sera bornée aux constructions faisant partie de la bibliothèque Pharos, présentée dans l'annexe B.

Traduction des politiques techniques

Deux types de politiques techniques responsables de l'exécution des composants existent dans la bibliothèque Pharos : les politiques techniques d'exécution périodique et les politiques techniques d'exécution unique. Comme indiqué dans la table 6.3, la traduction de ces deux concepts utilise les éléments du métamodèle *verif_exec* suivants : EmScenario, EmExecutionStep et EmExecutionPattern.

La traduction d'une de ces politiques techniques commence donc par la création d'un nouvel EmScenario, qui n'a aucun prédécesseur. Cette traduction crée également la première étape de ce scénario sous la forme d'un EmExecutionStep, qui servira donc d'état racine du scénario. Enfin, pour dicter l'activation de ce nouveau scénario, un EmExecutionPattern est associé à cet état racine.

Dans le cas d'une politique d'exécution périodique, le EmExecutionPattern portera les informations de période et de phase d'exécution. Un EmExecutionPattern représentant une politique d'exécution unique ne portera que l'information de phase.

La traduction d'une politique technique d'exécution périodique, telle que celle qui est associée au composant *controle*, est présentée par le schéma 6.8. La présence d'une politique technique d'exécution périodique implique que le composant auquel elle s'applique possède une méthode appelée *run*, dans le cas du composant *controle* cette méthode est visible dans le listing 6.2.

La transition vers la partie traduisant la spécification du comportement du composant se fait par la création d'un sous scénario *run* qui sera le seul successeur de l'état racine créé lors de la traduction de la politique technique.

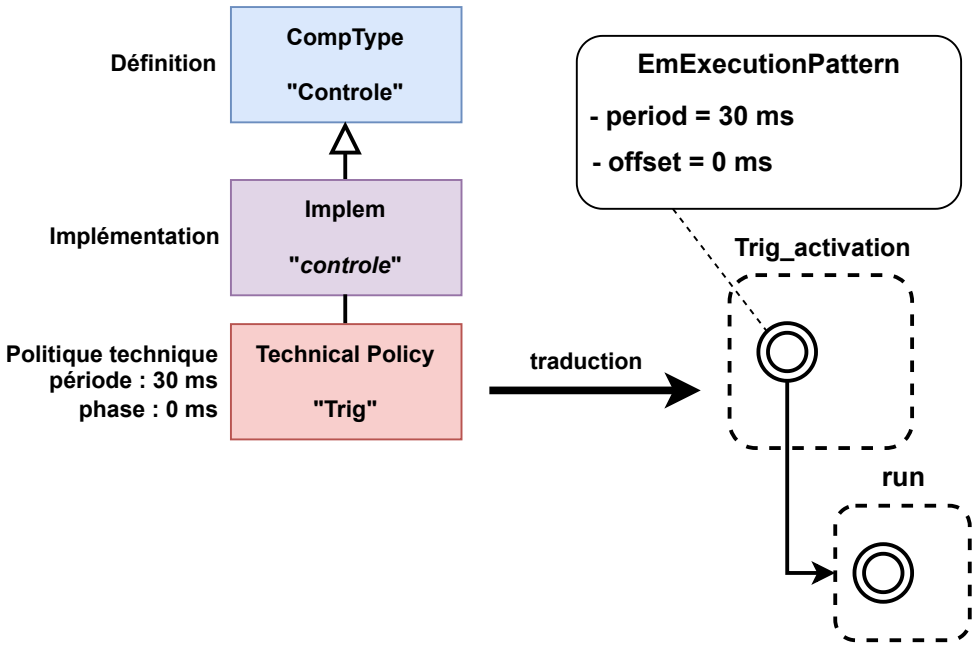


FIGURE 6.8 – Schéma présentant la traduction de la politique technique d'exécution périodique du composant *controle* vers le métamodèle *verif_exec*.

Traduction des comportement des composants

Cinq constructions permettent la spécification du comportement d'une méthode d'un composant : l'étape de calcul, l'appel de méthode d'un port, la séquence, l'alternative et la boucle.

Le comportement d'une méthode est traduit par un EmScenario qui contiendra les étapes qui la compose.

Une étape de calcul est traduite directement sous la forme d'un EmExecutionStep au sein du scénario. Les informations de meilleur et de pire temps d'exécution sont conservées.

Un appel de méthode d'un port, c'est-à-dire un appel sortant du composant, est traduit par un EmExecutionStep lié à un sous scénario. Ce sous scénario contiendra le comportement de la méthode appelée. Le paramètre *WaitForSubGroup* de l'EmExecutionStep indique que l'exécution de ce sous scénario doit être complétée avant de poursuivre l'exécution du scénario en cours. Cette traduction reproduit le comportement d'un appel de méthode classique dans lequel l'appelant doit attendre que l'appelé ait terminé.

Une séquence n'est pas traduite par un EmExecutionStep, mais par le fait que plusieurs EmExecutionStep sont chaînés les uns à la suite des autres.

La méthode *run* du composant *controle* contient des étapes de calcul et d'appel de méthodes qui forment une séquence. La figure 6.9 présente la traduction de cette méthode dans le métamodèle *verif_exec*. Suite au déclenchement du scénario représentant la méthode *run* par le scénario représentant la politique technique d'exécution périodique, les étapes de la séquence s'enchaînent. L'étape *read_data* fait appel à un sous scénario et attend la fin de son exécution avant de poursuivre l'exécution du scénario *run*. La dernière étape fait appel à la méthode *push* et déclenche l'activation du scénario représentant le connecteur qui se chargera de réaliser l'envoi de message.

Une alternative est traduite par un EmExecutionStep possédant plusieurs successeurs. Cet EmExecutionStep porte l'information que l'exécution du scénario en cours ne peut se poursuivre

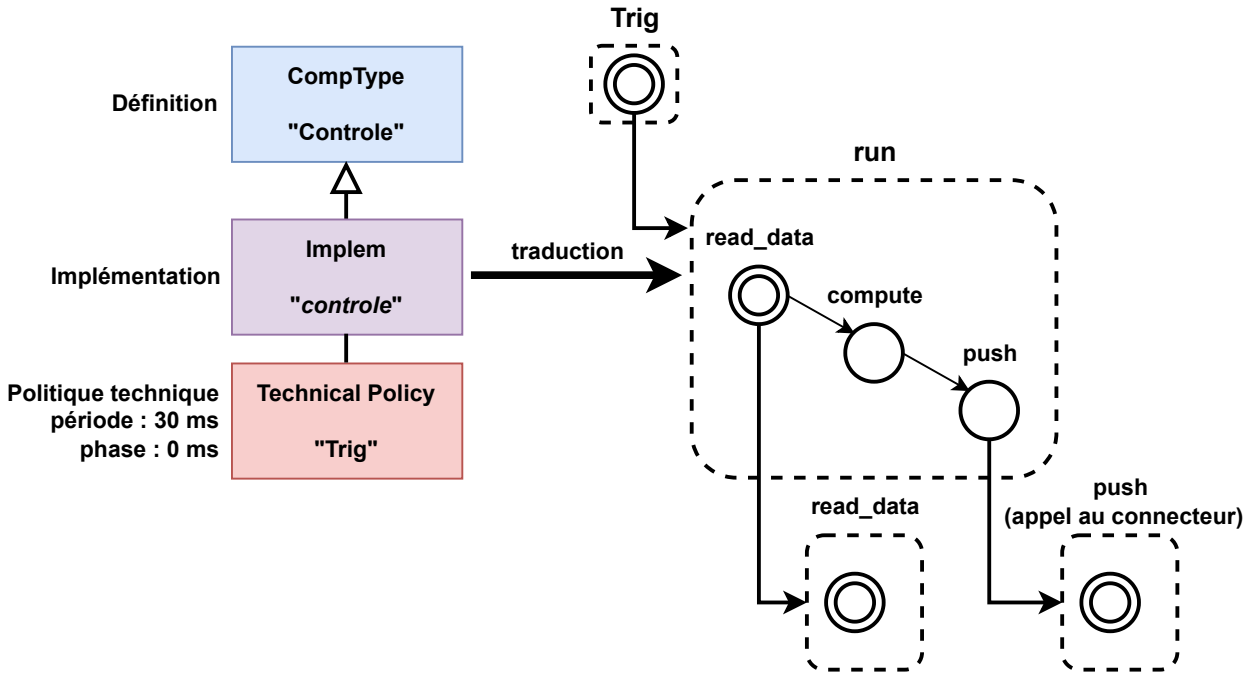


FIGURE 6.9 – Schéma présentant la traduction de la spécification de la méthode *run* du composant *controle* vers le métamodèle *verif_exec*.

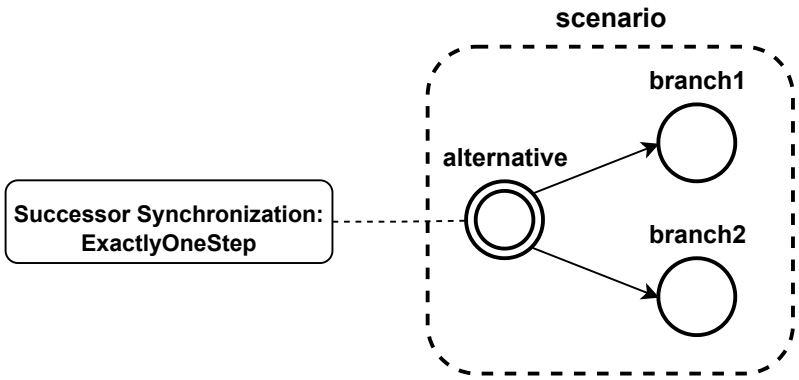


FIGURE 6.10 – Schéma présentant la traduction d'une alternative dans le métamodèle *verif_exec*.

que par un seul EmExecutionStep parmi ses successeurs. Dans la figure 6.10, cette information est représentée par la valeur *ExactlyOneStep* qui s'applique à la synchronisation des successeurs de l'étape racine du scénario.

Une boucle est traduite par un EmExecutionStep possédant un sous segment et deux informations : le nombre minimum et maximum d'itérations de la boucle. Le sous segment contient les EmExecutionStep constituant le comportement de la boucle. La boucle représentée dans la figure 6.11 contient un sous segment de trois étapes qui sera exécuté entre une et cinq fois.

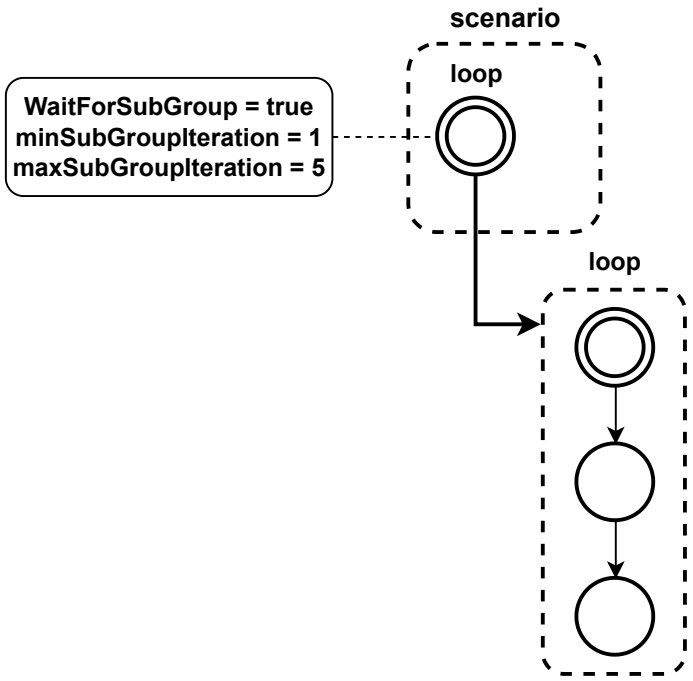


FIGURE 6.11 – Schéma présentant la traduction d'une boucle dans le métamodèle *verif_exec*.

Traduction des connecteurs

Les quatre connecteurs définis dans la bibliothèque Pharos, présentés dans l'annexe B sont traduits de façon similaire.

La traduction de l'appel à un connecteur commence par la création d'un scénario dont l'état racine est un EmExecutionStep qui représente l'emballage des données par la bibliothèque CBOR [CBO13]. Ces données sont ensuite envoyées par une *socket* UDP, ce qui est traduit par l'appel d'un sous scénario qui contient un EmCommStep. La réception des données est également traduite par un sous scénario qui les distribuera au composant destinataire. Enfin, la méthode en charge de la réception des données dans le composant destinataire est appelée par un dernier sous scénario.

Cet enchaînement de sous scénarios est présenté dans la figure 6.12, qui décrit la traduction de l'utilisation d'un connecteur engendré par l'appel de la méthode *push* du composant *controle*. Ce cas est relativement simple car il n'implique qu'une unique transmission ayant un unique composant destinataire. Dans ce cas, un unique EmCommStep, élément central de la traduction des connecteurs dans le métamodèle *verif_exec*, est créé.

Dans le cas général, l'appel à un connecteur peut engendrer la création de plusieurs EmCommStep, représentants donc plusieurs transmissions de flux de données sur le réseau. Cela se produit si des données doivent être transmises d'un composant vers plusieurs autres composants, déployés sur des nœuds différents. Ces multiples transmissions sont initiées par les EmExecutionStep du premier scénario, dont le paramètre *WaitForSubGroup* est à faux, ce qui signifie que ces étapes peuvent appeler leur sous scénario et ne pas attendre la fin de son exécution, représentant ainsi l'émission de plusieurs flux de données de manières asynchrone.

Il est également possible qu'un flux de données ait plusieurs composants destinataires déployés sur un même nœud, auquel cas une seule transmission, et donc un seul EmCommStep,

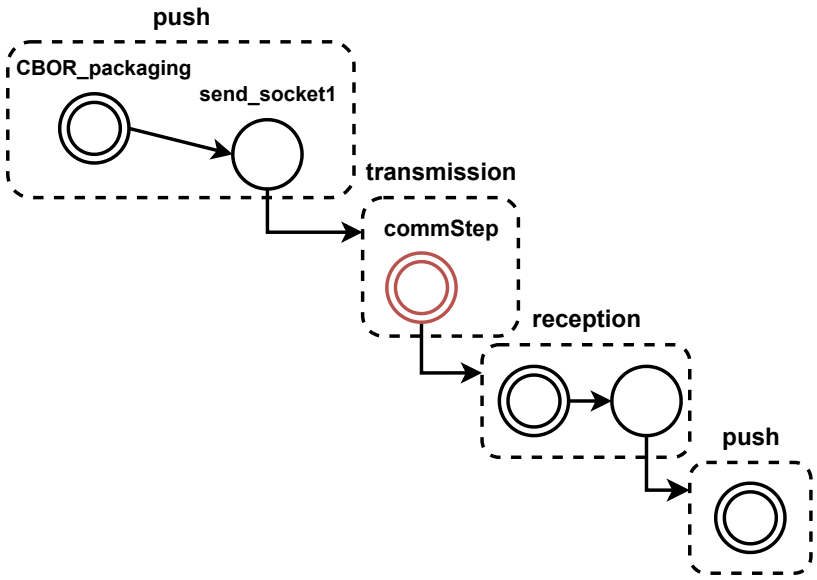


FIGURE 6.12 – Schéma présentant la traduction du connecteur utilisé par la méthode *push* du composant *controle* dans le métamodèle *verif_exec*.

est générée. La distribution des données aux différents composants destinataires aura lieu après la réception du flux de données sur le nœud, par une étape appelée *dispatch*.

La figure 6.13 présente ce cas général, dans lequel la traduction de l'utilisation d'un connecteur, que ce soit pour l'envoi de message ou pour de la donnée partagée (dont la définition est données dans l'annexe B), engendre la création de plusieurs EmCommStep et la distribution des données vers plusieurs composants destinataires sur chaque nœuds.

Les connecteurs disponibles dans la bibliothèque Pharos sont des connecteurs *unicast*. Dans l'hypothèse où des connecteurs *multicast* seraient également disponibles, il est tout à fait possible de les représenter dans le métamodèle *verif_exec*. Pour ce faire, les sous scénarios *transmission* présents sur la figure 6.13 contiendront des étapes supplémentaires au simple *EmCommStep* dont le rôle sera d'effectuer les transmissions vers les différents composants déployés sur des nœuds différents.

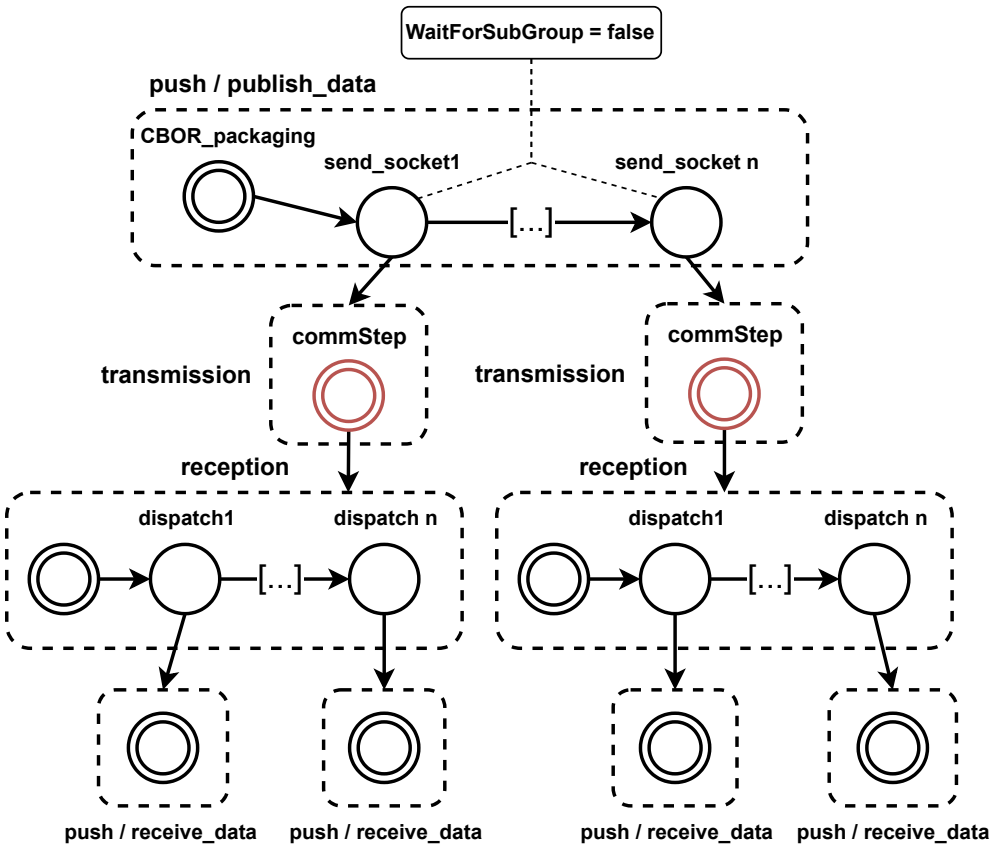


FIGURE 6.13 – Schéma présentant la traduction d'un connecteur dans le métamodèle *verif_exec* dans le cas général.

6.3 Génération automatique du modèles des flux de données

Une fois l'assemblage de la spécification du comportement des composants obtenu, il devient possible de faire les calculs nécessaires et d'extraire les valeurs des paramètres des flux de données.

Comme expliqué dans le chapitre 5, ces paramètres sont divisés en deux catégories : les paramètres intrinsèques aux flux de données (e.g. leur période et leur phase de transmission, la taille des données transmises, etc.), et les paramètres relevant des exigences système (e.g. les contraintes de latence et de gigue, le chemin emprunté, etc.). Ces derniers relèvent de la responsabilité de l'architecte système et ne peuvent pas être générés automatiquement à partir du modèle de l'architecture logicielle.

Dans cette section, nous présenterons la façon dont sont obtenus chacun des paramètres intrinsèques aux flux de données. Nous séparerons le calcul des valeurs des paramètres temporels du reste car ce sont ceux qui sont le plus difficile à obtenir.

6.3.1 Taille des données, classe de trafic, émetteur et destinataire

Tout comme dans le reste ce manuscrit, nous utiliserons notre système exemple (présenté par la figure 5.6) pour illustrer les explications. Nous nous concentrerons sur les deux flux de contrôle-commande échangés entre les terminaux *Controle* et *Moteur* dont les paramètres calculés sont présentés dans le listing 6.3.

```
[stream2]
 name = "consigne_conn_controle1_consigne_moteur_emtr_pe_to_moteur1"
 payload.value = 38
 payload.unit = "B"
 type = "Periodic_Stream"
 period.value = 30000
 period.unit = "us"
 talker = ["::model::topologie1::controle_to_pont2.controle_eth0"]
 listener = ["::model::topologie1::moteur_to_pont1.moteur_eth0"]
 pcp = ["7"]
 offset.value = 11002
 offset.unit = "us"

[stream3]
 name = "etat_conn_moteur1_etat_emtr_pe_to_controle1_etat_moteur"
 payload.value = 44
 payload.unit = "B"
 type = "Periodic_Stream"
 period.value = 30000
 period.unit = "us"
 talker = ["::model::topologie1::moteur_to_pont1.moteur_eth0"]
 listener = ["::model::topologie1::controle_to_pont2.controle_eth0"]
 pcp = ["7"]
 offset.value = 26004
 offset.unit = "us"
```

Listing 6.3 – Paramètres des flux de données *consigne* et *etat*.

Taille des données

Tous les types de données supportés par UCM ont une taille en mémoire connue, e.g. un entier 16 bits, une chaîne de caractères avec une longueur maximum. Dans le cas de notre système exemple, les connecteurs UCM utilisés sérialisent la charge utile grâce à la bibliothèque CBOR [CBO13] [17]. Cette sérialisation a comme objectif d'éviter à avoir à anticiper les problématiques d'alignement mémoire et de pouvoir calculer la taille des données de manière systématique.

Ces données sont transmises par les protocoles IP et UDP dont la taille des en-têtes représente respectivement 20 et 8 octets. La prise en compte de la taille de ces en-têtes sera faite par ailleurs, au moment du calcul de la configuration

Le standard UCM permet donc de calculer et d'extraire du modèle la taille des données qui seront échangées lors des communications réseau. Le génération automatique de l'armature logicielle servant de connecteur permet de garantir que la taille des données qui seront réellement échangées par l'application est la même que celle extraite pour le modèle des flux de données.

Dans le cas de notre exemple, la taille des données du flux échangé entre *Controle* et *Moteur* est de 38 octets et celle du flux échangé entre *Moteur* et *Controle* est de 44 octets. Il est important de préciser qu'une trame Ethernet ne peut avoir une taille inférieure à 64 octets et que par conséquent une charge utile de taille inférieure à cette valeur sera arrondie pour l'atteindre. Nous conservons les valeurs telles qu'elles sont calculées pour des raisons de clarté

17. La sérialisation CBOR s'apparente à une sérialisation en JSON utilisant un format binaire au lieu de chaînes de caractères.

et de documentation.

Classe de trafic

Le paramètre de classe de trafic est, dans la pratique, associé à un niveau de priorité spécifique. Notre approche ayant pour but de concevoir et de configurer des réseaux TSN, nous pouvons utiliser les 8 niveaux de priorités prévus par les standards. Ces niveaux de priorités vont de 0, niveau de priorité la plus faible – qui peut aussi être lu « niveau le moins prioritaire » – à 7, niveau de priorité la plus élevée.

La notion de priorité est incluse dans les connecteurs d'UCM, depuis lesquels nous pouvons donc extraire cette information. Les deux flux de données qui nous servent d'exemple dans cette section ont le même niveau de priorité, le plus élevé, car ils appartiennent à la classe de trafic contrôle-commande. Ce paramètre est appelé *Priority Code Point* (PCP) dans la terminologie TSN, qui est utlisée dans le listing 6.3.

Émetteur et destinataire

Comme présenté dans le chapitre 3, les modèles UCM contiennent un ou plusieurs plans d'allocation, qui spécifient quels composants sont déployés sur quel nœud. Bien que les composants soient reliés entre eux par les connecteurs, au niveau de la modélisation de l'architecture logicielle en UCM la topologie du réseau n'est pas connue. Il n'est donc pas possible d'extraire le chemin emprunté par les flux de données de ce modèle mais il est possible d'en extraire les informations d'émetteur et de destinataire, par le biais des interfaces réseau des terminaux sur lesquels les composants sont déployés.

Comme le montre le listing 6.3, ces informations sont bien présentes et indiquent bien que les deux flux de données représentés sont échangés entre les terminaux *Controle* et *Moteur*.

6.3.2 Période de transmission

Afin d'extraire et de calculer les valeurs des paramètres intrinsèques de flux de données pour produire la configuration du réseau et surtout des mécanismes d'ordonnancement, nous devons considérer le pire cas. Dans cette perspective, le pire cas est déterminé par les trois critères suivants :

— le plus grand volume de données ;

— le plus grand nombre de flux de données ;

— la période de transmission la plus faible (c'est-à-dire la fréquence de transmission la plus élevée).

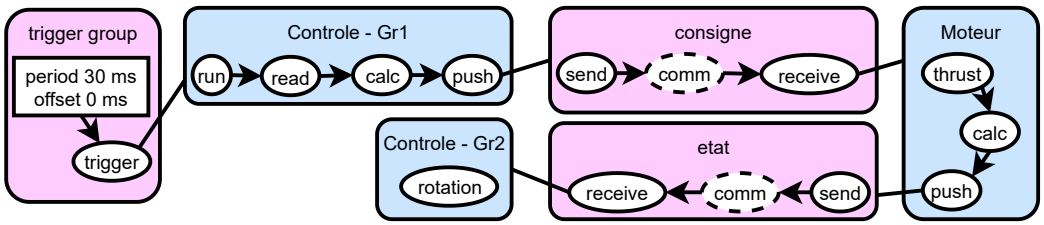


FIGURE 6.14 – Schéma de l'assemblage du comportement des composants déployés sur les terminaux *Controle* et *Moteur*, engendrant les deux flux de données de contrôle-commande.

La nature des flux de données dépend de l'entité qui les produit, elle peut être périodique ou sporadique. La figure 6.14 présente l'assemblage du comportement des composants *Controle* et *Moteur* tel que traduit dans le métamodèle *verif_exec*. Cet assemblage contient deux étapes de communication (EmCommStep), appelées *comm*, une pour le flux de données *consigne* et une pour le flux de données *etat*.

La première de ces étapes de communication représente la transmission du flux de données *consigne*. Ce flux de données est engendré par le composant *Controle* auquel est associé une politique technique d'exécution périodique qui est responsable de son déclenchement. Ce cas est le plus simple à partir duquel extraire et calculer les informations concernant le flux de données : dans ce cas, le flux de données *consigne* est de nature périodique et sa période de transmission correspond à celle du composant qui le produit, c'est-à-dire 30 ms.

La deuxième étape de communication présente dans la figure 6.14 représente la transmission du flux de données *etat*. Ce flux de données est engendré par le composant *Moteur* auquel aucune politique technique d'exécution périodique n'est associée. Seule une politique technique d'exécution passive est associée à ce composant, ce qui implique qu'il ne peut pas se déclencher par lui-même. Il n'est donc pas possible d'extraire directement les informations de nature et de période de transmission du flux de données *etat*.

Dans ce deuxième cas contenant une dépendance entre tâches, il est nécessaire de remonter à la cause du déclenchement du composant à exécution passive dans l'assemblage du comportement. Le déclenchement du composant *Moteur* se fait sur la réception du flux de données *consigne* produit par le composant *Controle*. La nature et la période de transmission du flux de données *etat* sont donc héritées de ce flux : c'est un flux de nature périodique avec une période de transmission de 30 ms.

Enfin, le troisième et dernier cas est celui des composants n'ayant pas de politique technique de déclenchement périodique et dont le déclenchement n'est pas la conséquence de l'exécution d'un autre composant, ce qui est typiquement le cas d'un composant dont l'exécution dépend d'un stimulus venant de l'extérieur du système. Ce cas mène à la création de flux de données de nature sporadique et sa période de transmission correspond à l'intervalle d'interarrivé du composant. Cet intervalle indique l'intervalle de temps minimum entre deux exécutions du composant, l'utiliser comme période de transmission correspond donc à nos critères du pire cas.

6.3.3 Phase de transmission

Importance de la phase de transmission

Le dernier paramètre dont nous pouvons calculer la valeur est la phase de transmission des flux de données. La phase de transmission d'un flux de données désigne le décalage entre le démarrage du système et la première transmission du flux de données. Ce paramètre peut avoir un impact important sur les performances du réseau, voire invalider la configuration des mécanismes d'ordonnancement.

Comme le montre la figure 6.15, une erreur dans le calcul de la phase de transmission utilisée dans le modèle d'entrée du générateur de configuration peut avoir différents effets suivant si cette valeur est sous-estimée ou surestimée. En effet, cette valeur étant utilisée par le générateur de configuration pour synthétiser une configuration des mécanismes d'ordonnancement, si cette valeur ne correspond pas à la réalité cette configuration peut être entièrement invalidée.

Dans le cas où cette valeur est surestimée – c'est-à-dire que la trame réelle est transmise en avance par rapport à la phase de transmission présente dans le modèle, ce qui correspond au premier cas de la figure 6.15 – le risque est une augmentation de la latence du flux de données.

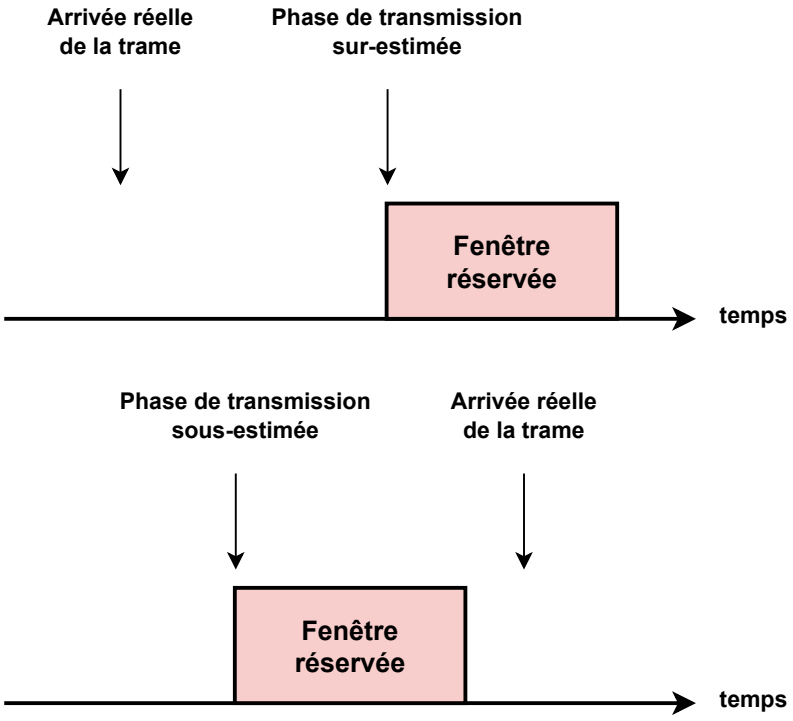


FIGURE 6.15 – Schéma de l'impact d'une phase de transmission mal calculée.

Cette augmentation de latence est due au fait que la trame étant transmise plus tôt que prévu, elle est réceptionnée plus tôt par le pont par lequel son chemin passe, or la génération de la configuration étant basée sur la valeur de la phase de transmission présente dans le modèle, la fenêtre protégée n'est pas encore active et la trame devra attendre.

Cette erreur de calcul de la valeur de la phase de transmission n'invalide pas la configuration du réseau car elle n'empêche pas le respect des échéances des flux de données mais elle dégrade les performances du réseau en augmentant la latence.

Dans le deuxième cas présenté par la figure 6.15, la valeur de la phase de transmission est sous-estimée – c'est-à-dire que la trame réelle est transmise en retard par rapport à la phase de transmission présente dans le modèle – et les conséquences de cette erreur peuvent être plus grave.

Une erreur de ce type conduira la trame à rater la fenêtre de transmission qui lui est réservée par le TAS. Rater une fenêtre réservée pour une trame peut avoir plusieurs conséquences : s'il y a une autre fenêtre réservée pour le même niveau de priorité dans le cycle TAS, la trame sera transmise pendant cette autre fenêtre, sa latence sera donc augmentée et elle gênera les trames censées être transmises pendant cette autre fenêtre, répétant ce problème, s'il n'y a pas d'autre fenêtre réservée pour le niveau de priorité de la trame, elle devra attendre le prochain cycle TAS pour pouvoir être transmise par le pont.

Dans les deux cas, lorsqu'une trame rate sa fenêtre réservée il est probable que les flux de données impactés ratent leurs échéances respectives. En effet, la durée d'un cycle TAS est généralement plus longue que l'échéance d'un flux de données. Par exemple, dans le cas des flux de données de contrôle-commande *consigne* et *etat*, leurs échéances sont fixées à 10 ms alors que la durée des cycles TAS aux ports des ponts du réseau est de 30 ms.

Calcul de la phase de transmission

L'assemblage de la spécification du comportement des composants produisant les flux de données nous permet de calculer la valeur de la phase de transmission de chaque flux. Tout comme pour la période de transmission, cette valeur peut dépendre du comportement d'autres composants du système et nous exprimons son calcul de la manière suivante :

$$\phi_f = W_{prec} + \phi_{decl} + dl_{decl} \tag{6.1}$$

L'équation 6.1 permet de calculer la phase de transmission du flux de données *f*.

Le premier terme de cette équation désigne la somme des temps d'exécution dans le pire cas[18] des étapes de calcul qui précèdent la transmission du flux de données *f*. L'assemblage de la spécification du comportement nous permet de connaître le meilleur et le pire temps d'exécution de chacune des étapes, comme expliqué dans la section 6.2. Étant donné nos critères d'identification du pire cas à considérer pour la production de configuration, nous utilisons le pire temps d'exécution possible pour ce calcul. Si l'étape produisant le flux de données f est la j-ème étape de l'assemblage du comportement, alors on peut obtenir la valeur de ce terme par l'équation suivante :

$$W_{prec} = \sum_{i<j}^{0} WCET_i \tag{6.2}$$

Le deuxième terme de l'équation 6.1 désigne la phase de la cause du déclenchement de l'exécution du composant produisant le flux de données *f*. Tout comme pour le calcul de la valeur de la période de transmission, ce déclenchement peut avoir différentes causes : une politique technique d'exécution périodique, la réception d'un message pour un composant ayant une politique technique d'exécution passive ou un stimulus extérieur.

Un stimulus extérieur étant hors de la portée du modèle, nous considérons qu'il arrive instantanément au démarrage du système et que sa phase est donc nulle. Dans le cas de l'exécution déclenchée par la réception d'un message, la valeur de ce terme est égale à la phase de transmission du flux de données transmettant ce message. Dans le cas d'une politique technique d'exécution périodique, la valeur de ce paramètre est extraite de l'assemblage du comportement des composants de la même façon que la période, puisqu'il contient également cette information.

Le dernier terme de l'équation 6.1 n'est utile que dans le cas d'un flux de données produit par un composant auquel est associée une politique technique d'exécution passive et dont le déclenchement de l'exécution dépend de la réception d'un message. C'est par exemple le cas du flux de données *etat*.

Dans ce cas, il faut considérer la phase de transmission du flux de données causant le déclenchement mais également son temps de latence. Comme nous considérons le pire cas possible, nous devons considérer que le flux de données causant le déclenchement du composant produisant le flux de données f arrive le plus tard possible, c'est-à-dire au moment de son échéance.

Le listing 6.4 contient la spécification du comportement du composant *Controle*, que nous allons utiliser pour calculer la phase de transmission du flux de données qu'il produit : *consigne*. Cette spécification contient deux étapes de calcul : *calcul* et *calcul2*, dont le temps d'exécution dans le pire cas est respectivement de 1 ms et 10 ms. Ces informations nous sont données par des annotations portées par ces étapes de calcul.

L'étape de communication produisant le flux de données *consigne* étant créée par le connecteur appelé par la dernière étape de cette spécification – celle de la méthode *push_consigne* –

18. Worst Case Execution Time (WCET)

```
<detailed lang="::ucm_lang::cpp::CPP11_typed" name="controle1" type="::
    ↪ model::comps::controle">
 [...]
 <methScen meth="run" name="run" port="exec_periodique_activation">
  <seq name="seq">
   <call meth="read_data" name="read_mission" port="
   ↪ consigne_mission_rdr_pe"/>
   <comput name="calcul">
    <annotation def="::sigil_ann::tap::ta_step_timing">
     <config def="bcet" value="{1,ms}"/>
     <config def="wcet" value="{1,ms}"/>
    </annotation>
   </comput>
   <call meth="read_data" name="read_etat" port="etat_moteur_rdr_pe"/>
   <comput name="calcul2">
    <annotation def="::sigil_ann::tap::ta_step_timing">
     <config def="bcet" value="{8,ms}"/>
     <config def="wcet" value="{10,ms}"/>
    </annotation>
   </comput>
   <call meth="push" name="push_consigne" port="consigne_moteur_emtr_pe"
   ↪ />
  </seq>
 </methScen>
</detailed>
```

Listing 6.4 – Spécification du comportement du composant *Controle* utilisée pour calculer la phase de transmission du flux de données *consigne*.

le calcul de la valeur du premier terme de l'équation 6.1 donne 11000 µs, nous devons ajouter à cette valeur le temps d'appel à la bibliothèque de *socket* afin d'effectuer la communication réseau, cela représente 2 µs. Cette valeur étant de la responsabilité du connecteur, et pas du composant, elle n'apparaît pas dans la spécification. Elle est ajoutée automatiquement lors de la création de l'assemblage de la spécification du comportement et sa valeur est spécifiée dans la bibliothèque de plateforme (voir le chapitre 3). Nous négligeons le temps pris par les deux étapes *read_mission* et *read_data* car ce sont de simples lectures de valeurs.

Le composant *Controle* étant déclenché par une politique technique d'exécution périodique, nous ajoutons maintenant la valeur de la phase qui lui est associée, dont la valeur est nulle dans ce cas. Enfin, le dernier terme de l'équation est nul également car la cause du déclenchement de ce composant est une politique technique et ces entités n'ont pas d'échéance.

La valeur finale de la phase de transmission du flux de données *consigne* est de 11002 µs, comme présenté dans le listing 6.3.

Le listing 6.5 contient la spécification du comportement du composant *Moteur*, que nous allons utiliser pour calculer la phase de transmission du flux de données qu'il produit : *etat*. Cette spécification contient une étape de calcul dont le temps d'exécution dans le pire cas est de 5 ms.

L'étape de communication produisant le flux de données *etat* étant créée par le connecteur appelé par la dernière étape de cette spécification – celle de la méthode *push_etat* – le calcul de la valeur du premier terme de l'équation 6.1 donne 5000 µs. Tout comme pour le flux *consigne*,

```
<detailed lang="::ucm_lang::cpp::CPP11_typed" name="moteur1" type="::
    ↪ model::comps::moteur">
 [...]
 <methScen meth="push" name="push" port="consigne_rcvr_pe">
  <seq name="seq">
   <comput name="calcul">
    <annotation def="::sigil_ann::tap::ta_step_timing">
     <config def="bcet" value="{5,ms}"/>
     <config def="wcet" value="{5,ms}"/>
    </annotation>
   </comput>
   <call meth="push" name="push_etat" port="etat_emtr_pe"/>
  </seq>
 </methScen>
</detailed>
```

Listing 6.5 – Spécification du comportement du composant *Moteur* utilisée pour calculer la phase de transmission du flux de données *etat*.

nous ajoutons 2 µs à cette valeur.

Le composant *Moteur* étant associé à une politique technique d'exécution passive, la valeur du deuxième terme de l'équation est égale à la phase de transmission de la cause du déclenchement du composant, c'est-à-dire la phase de transmission du flux *consigne*.

Enfin, le dernier terme a pour valeur l'échéance du flux de données *consigne*, c'est-à-dire 10 ms. La valeur finale de la phase de transmission du flux de données *etat* est de 5002 + 11002 + 10000 = 26004 µs.

6.3.4 Génération des flux de données dans les cas particuliers

Comme expliqué précédemment, les modèles des flux de données sont générés lorsqu'une étape de communication est rencontrée lors du parcours de l'assemblage du comportement des composants (qui inclut également les connecteurs et les politiques techniques). La spécification des comportements peut contenir des séquences d'étapes mais il est aussi possible d'utiliser deux autres constructions : les alternatives et les boucles.

Les alternatives sont des embranchements parmi lesquels seules une branche peut être empruntée à chaque exécution. Les boucles sont une répétition d'un ensemble d'étapes dont les nombre minimum et maximum d'itérations sont spécifiés.

Afin de pouvoir synthétiser une configuration permettant de respecter les exigences du système dans tous les cas, l'utilisation de ces constructions entraîne l'utilisation de règles particulières lors de la génération du modèle des flux de données.

Le cas des alternatives

Les modèles des flux de données engendrés par des étapes situées dans les branches d'une alternative sont générés de la même manière que tout autre flux de données. Néanmoins, il est impératif de prévoir tous les embranchements possible pour pouvoir synthétiser une configuration correcte.

La figure 6.16 présente une alternative telle qu'il est possible d'en utiliser dans la spécification du comportement d'un composant. On remarque qu'il est possible de spécifier des appels de

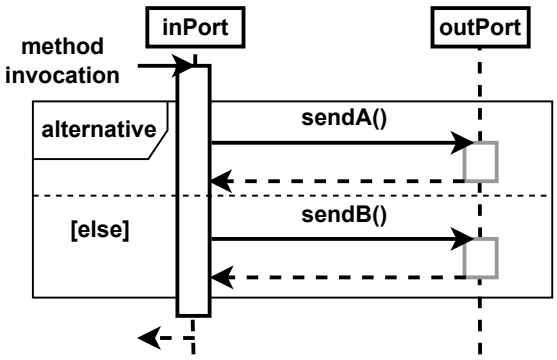


FIGURE 6.16 – Diagramme de séquence d'une alternative comportant des communications réseau.

méthode engendrant des communications réseau dans chacune des branches d'une alternative, pouvant être précédées, ou pas, d'étapes de calcul et le nombre de branches n'étant pas limité.

Afin de pouvoir garantir le respect des exigences systèmes, nous devons synthétiser une configuration valide peu importe quelle branche est sélectionnée à chaque exécution. Dans le cas présenté ici, cela implique de générer deux modèles de flux de données : un pour *sendA()* et un pour *sendB()*. La configuration synthétisée sera donc prévue pour garantir le respect des exigences pour ces deux flux de données en même temps, même si une seule branche sera exécutée à chaque déclenchement du composant.

Il n'est pas souhaitable de regrouper tous les flux de données engendrés par les étapes présentes dans les différentes branches d'une alternative afin de ne créer qu'une unique fenêtre protégée adaptée à toutes les branches car elle peuvent être précédées par des étapes de calcul qui entraîneraient des phases de transmission très différentes pour chaque branche et donc une fenêtre qui pourrait accaparer toutes les ressources du réseau.

Le cas des boucles

Comme pour les alternatives, les modèles des flux de données engendrés par des étapes situées dans le corps d'une boucle sont générés de la même manière que tout autre flux de données. La multiplication des itérations du corps d'une boucle entraîne néanmoins une génération particulière de ces modèles, toujours pour garantir le respect des exigences du système dans tous les cas.

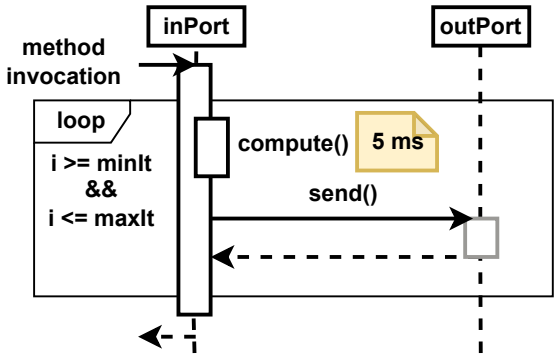


FIGURE 6.17 – Diagramme de séquence d'une boucle comportant des communications réseau.

La figure 6.17 présente une boucle telle qu'il est possible d'en utiliser dans la spécification du comportement d'un composant. On remarque qu'il est nécessaire de spécifier le nombre minimum et maximum d'itération effectuées par cette boucle.

Afin de pouvoir garantir le respect des exigences systèmes dans le pire cas, il est nécessaire de considérer que le nombre d'itérations sera toujours égal au maximum. Cela implique de générer un modèle représentant un flux de données pour chaque étape du corps de la boucle engendrant une communication réseau pour chaque itérations de la boucle. Si le corps d'une boucle comporte deux étapes engendrant des communications réseau et que son nombre maximum d'itérations est fixé à cinq, alors le modèle des flux de données correspondant contiendra dix flux de données.

Le corps d'une boucle peut également contenir des étapes de calcul, la phase de transmission de chaque flux de données générés devra donc prendre en considération le temps d'exécution dans le pire cas de ces étapes et le multiplier par l'indice de l'itération engendrant le flux de données. La figure 6.17 présente une boucle dont le corps contient une étape de calcul dont le pire temps d'exécution est de 5 ms et une étape engendrant un flux de données. Pour la première itération, la phase de transmission du flux de données comptabilisera une fois de temps (5 ms), pour la deuxième deux fois (10 ms), pour la troisième trois fois (15 ms), etc.

6.4 Discussion et conclusion

Dans ce chapitre, nous avons présenté notre approche de modélisation automatique des flux de données. Le modèle des flux de données du système étant la base de ce qui sert à générer la configuration du réseau, et notamment à synthétiser la configuration des mécanismes d'ordonnancement, avoir un processus de modélisation automatique est particulièrement précieux.

En effet, cette étape de modélisation peut s'avérer difficile, car l'obtention des valeurs des paramètres intrinsèques des flux de données (volume des données, période et phase de transmission, etc.) n'est pas évidente dans tous les cas et il est difficile de ne pas commettre des erreurs en les calculant sans assistance.

En plus d'éviter les erreurs humaines lors de la modélisation des flux de données, notre approche de modélisation automatique permet d'assurer la cohérence entre le modèle des flux de données et les flux de données réels qui seront produits par l'application une fois le système mis en marche.

Pour ce faire, nous lions la modélisation de l'architecture logicielle en UCM, comme présentée dans le chapitre 3, et la modélisation réseau, comme présentée dans le chapitre 5. Nous utilisons cette modélisation de l'architecture pour en extraire les valeurs des paramètres intrinsèques des flux de données. La garantie de cohérence entre ce modèle des flux de données obtenu automatiquement et les flux de données réels est assurée par l'utilisation de la modélisation de l'architecture logicielle à la fois pour générer le modèle des flux et le code de l'armature logicielle responsable, entre autre, de gérer les communications réseau effectuées par les applications.

Afin de pouvoir extraire les informations nécessaires à la modélisation automatique des flux de données, le modèle de l'architecture logicielle doit être enrichi d'une spécification du comportement des composants. Cette spécification de comportement est inspirée des diagrammes de séquences d'UML et peut contenir un ensemble d'étapes de calcul et d'appel de méthode – pouvant potentiellement engendrer des communications réseau – en séquence, en boucle ou dans les branches d'une alternative.

Le comportement des composants ne suffisant pas à décrire entièrement le comportement de l'application, il est nécessaire de le regrouper avec les politiques techniques et les connecteurs pour avoir accès à l'ensemble des informations nécessaires à la modélisation des flux. C'est le rôle

du métamodèle intermédiaire que nous utilisons, qui permet de créer un assemblage regroupant le comportement des composants, des politiques techniques et des connecteurs afin de décrire le comportement de l'application entièrement. Le métamodèle *verif_exec* est donc une contribution de cette thèse.

Une fois cet assemblage obtenu, nous pouvons extraire et calculer les valeurs des paramètres servant à la modélisation des flux de données. Les valeurs des paramètres suivants peuvent être obtenues : la taille des données, l'émetteur et le destinataire, le niveau de priorité, la période et la phase de transmission au niveau des terminaux.

Cette approche permet de garantir la cohérence entre les flux de données modélisés et les flux de données réels mais elle permet également de vérifier que l'implémentation des composants respecte bien la spécification de leur comportement. Comme présenté dans [SV19], le modèle de l'architecture logicielle permet également de générer des oracles de test qui comparent les traces d'exécution de l'application avec la spécification de son comportement sur les mêmes bases que celles utilisée pour générer automatiquement le modèle des flux de données.

Tous les problèmes liés à la modélisation automatique des flux de données ne sont pas résolus par cette contribution. Nous faisons dans notre approche l'hypothèse que les temps d'exécution dans le pire cas des étapes de calcul sont connus, or leur obtention n'est pas forcément simple. Des travaux sur l'évaluation de ces pire temps d'exécution existent mais sont hors du périmètre de cette thèse. Dans [WEE+08], une revue des méthodes et des outils permettant d'évaluer le temps d'exécution dans le pire cas est présentée.

La façon dont nous proposons de calculer la phase de transmission d'un flux de données peut être pessimiste dans certains cas. Il peut y avoir un grand écart entre l'échéance d'un flux de données et sa latence de bout en bout dans le pire cas, qu'il est possible d'obtenir grâce aux résultats d'outils d'analyse, la trame pouvant arriver bien avant son échéance. Dans un cas tel que celui du flux *etat*, qui est produit par un composant ne se déclenchant pas de lui-même, nous utilisons l'échéance du flux de données produisant ce déclenchement pour calculer la phase de transmission. Cela mène à la première situation de la figure 6.15, puisque la configuration des mécanismes d'ordonnancement est calculée à partir de la phase de transmission (qui considère donc l'échéance et non la latence dans le pire cas [19]), et dégrade donc les performances du réseau (en augmentant la latence des flux de données les plus critiques) tout en garantissant le respect des contraintes temps réel.

Cette limite peut encourager l'architecte du système à modifier la spécification du système. Notre approche pouvant être utilisée dans le cadre d'une démarche itérative, il est possible de modéliser le système une première fois, de générer la configuration et d'analyser les résultats des simulations et des outils d'analyse avant de faire évoluer le modèle et de recommencer. Cela permet de réduire les échéances et de limiter l'impact de ce phénomène. Une autre possibilité est de réduire l'utilisation de composants ne se déclenchant pas par eux-mêmes afin d'éliminer l'utilisation de l'échéance dans le calcul des phases de transmission.

Une autre limite de notre approche est la production du modèle des flux de données à partir d'une spécification du comportement dans laquelle de nombreuses boucles et alternatives sont utilisées. Ces constructions, lorsqu'elles contiennent des étapes produisant des flux de données, conduisent elles aussi à un certain pessimisme et donc a un gâchis d'une partie des ressources du réseau . Il est nécessaire de produire une configuration pouvant s'adapter à tous les cas possibles

19. Il n'est pas possible d'utiliser la latence dans le pire cas pour ce calcul car il faut déjà avoir généré la configuration du réseau pour obtenir cette valeur. Toutefois, étant donné que notre approche permet la mise en place d'un processus de conception itératif, il est possible d'utiliser cette valeur après avoir généré la configuration du réseau une première fois afin d'utiliser un outil d'analyse calculant la latence dans le pire cas puis de mettre à jour le modèle en modifiant la valeur de l'échéance.

lors de l'exécution. Cela mène à produire une configuration garantissant le respect des échéances de tous les flux de données alors qu'une partie d'entre eux ne seront pas systématiquement produits (c'est le rôle d'une alternative). Ce pessimisme est d'autant plus grand lorsque des boucles et des alternatives sont utilisées de façon imbriquée.

Il peut donc être bénéfique pour les performances du système de limiter l'utilisation de ces constructions. Privilégier l'utilisation de transmissions systématiques – par opposition aux transmissions dépendantes de la réception d'une autre transmission – permet de palier ce problème, quitte à augmenter la taille des données transmises pour pouvoir accueillir les données qui seraient transmises par les flux de données des différentes branches des alternatives.

Une dernière limite à notre approche est l'hypothèse que nous faisons concernant l'ordonnancement des tâches des différents terminaux du système. Ce problème étant hors du cadre de ces travaux de thèse, nous considérons que l'ordonnancement des terminaux sur lesquels sont exécutées les applications utilisant le système est parfaitement synchronisé. L'implication de cette hypothèse est que le déclenchement des composants respecte parfaitement leur période et leur phase d'exécution. Ce problème est traité par ailleurs dans des travaux comme [Kop11].

Des travaux futurs pourraient avoir pour objectif de lever cette hypothèse et d'intégrer l'ordonnancement des tâches au modèle du système afin de le prendre en considération dans la génération automatique du modèle des flux de données. Une autre possibilité de travaux futurs est de ne plus se limiter à l'utilisation du temps d'exécution dans le pire cas des étapes de calculs présentes dans la spécification du comportement des composants. Utiliser également le meilleur temps d'exécution pourrait permettre une réduction du pessimisme dans le calcul des phases de transmission des flux de données, potentiellement dans le cadre d'une approche de calcul probabiliste, prenant en compte les bornes inférieures et supérieures du pire temps d'exécution.

Une fois le modèle des flux de données produit et sa cohérence avec les flux de données réels qui seront échangés sur le système assurée, il devient possible de l'utiliser pour synthétiser la configuration des mécanismes d'ordonnancement de TSN puis pour générer les modèles utilisés par les simulateurs et les outils d'analyse qui serviront à valider la configuration du réseau produite.

Chapitre 7

Utilisation combinée du Time Aware Shaper et du Credit-Based Shaper

Dans certains cas, lors de la conception du système, il peut y avoir un intérêt à utiliser plusieurs mécanismes d'ordonnancement différents de façon conjointe. Certains systèmes font cohabiter des types de trafic très différents, de part le volume de données échangés, la fréquence de transmission ou encore la marge disponible pour respecter les contraintes temps réel, venant de la distinction entre les contraintes temps réel dures et souples.

L'utilisation de mécanismes d'ordonnancement différents peut alors être intéressante afin de pouvoir gérer certains flux de données avec un mécanisme d'ordonnancement et d'autres flux de données avec un autre. Cette façon de faire permet de tirer partie des avantages de chaque mécanisme d'ordonnancement tout en évitant leurs inconvénients – par exemple le sur dimensionnement de la quantité de bande passante allouée à une classe de trafic donnée – en leur attribuant les flux de données pour lesquels ils sont les plus adaptés.

Pour notre approche, les deux mécanismes d'ordonnancement les plus intéressant, car ce sont les plus adaptés aux types de trafic circulant sur les réseaux des systèmes que nous concevons, sont le *Time Aware Shaper* (TAS) et le *Credit-Based Shaper* (CBS). Le TAS est le plus adapté aux flux de données de type contrôle-commande, des flux généralement périodique et dont les contraintes sont très exigeantes pour le réseau. Le CBS est le plus adapté aux flux de données de type multimédia, des flux généralement très volumineux et pour lesquels le réseau dispose de plus de marge pour respecter leurs contraintes temps réel, le plus souvent parce que leur échéance est bien plus grande que le temps de transmission de leurs trames.

Il existe, pour ces deux mécanismes d'ordonnancement, des travaux permettant la synthèse de leur configuration lorsqu'ils sont utilisés séparément. Comme expliqué précédemment, il est également possible d'utiliser ces deux mécanismes d'ordonnancement de façon conjointe. Dans ce cas, il y aura une interaction entre les deux qui modifie leur comportement et qui nécessite donc une configuration différente de celle mise en place lorsque ces mécanismes sont utilisés séparément. Cette différence de configuration va, dans notre cas d'utilisation du TAS conjointement au CBS, impacter ce dernier. En effet, l'interaction résultante de l'utilisation conjointe de ces deux mécanismes d'ordonnancement n'affecte que le CBS.

Il est donc nécessaire de proposer une méthode permettant de produire la configuration du CBS lorsqu'il est utilisé conjointement au TAS. Cette méthode doit être capable de prendre en considération la présence du TAS dans la production de la configuration du CBS.

7.1 Pourquoi TAS et CBS

Le TAS et le CBS sont les deux mécanismes d'ordonnancement les plus couramment utilisés. Ce sont aussi les deux mécanismes d'ordonnancement les plus adaptés au type de systèmes que nous concevons et qui répondent le mieux aux contraintes industrielles que nous avons pu rencontrer au cours de cette thèse.

Leur utilisation conjointe nous permet de nous adapter aux différents types de trafic présents dans les systèmes que nous concevons. Le TAS et le CBS possèdent leurs propres spécificités et l'emploi d'un mécanisme d'ordonnancement adapté aux contraintes de chaque flux de données peut permettre d'éviter le gâchis d'une partie des ressources du réseau.

7.1.1 Les spécificités du TAS

Comme présenté dans le chapitre 2, le TAS est un mécanisme d'ordonnancement dont l'objectif est de supprimer la possibilité que le trafic critique soit perturbé par d'autres types de trafic. La division temporelle des transmissions permet d'atteindre cet objectif et de garantir que les contraintes temps réel des flux de données critiques, gérés par le TAS, seront toujours respectées, peu importe l'état du reste du trafic présent sur le réseau.

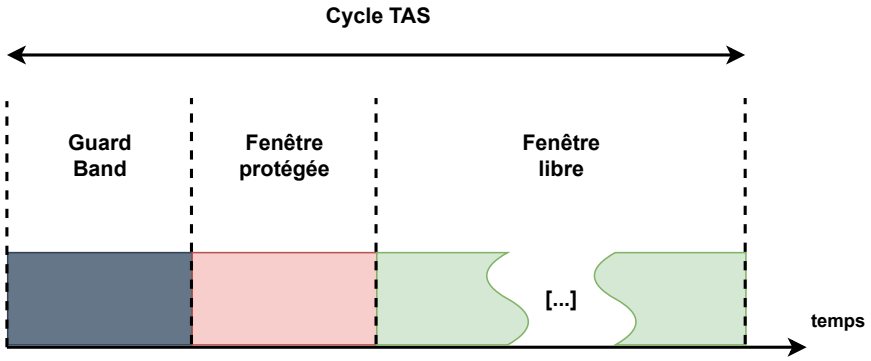


FIGURE 7.1 – Chronogramme typique d'un cycle TAS.

La figure 7.1 présente une configuration typique d'un cycle TAS – qui va donc se répéter indéfiniment – qui comporte, dans l'ordre du chronogramme, une fenêtre de *guard band* garantissant que la fenêtre suivante ne pourra pas être perturbée, une fenêtre protégée ne permettant qu'au trafic critique d'être transmis, et enfin une fenêtre non protégée permettant au reste du trafic d'être transmis.

Comme le montre cette figure, l'utilisation du TAS entraîne la définition de fenêtres de *guard band* pendant lesquelles aucune nouvelle transmission ne peut débuter (mais une trame ayant commencé sa transmission avant le début d'une fenêtre de *guard band* peut terminer sa transmission pendant celle-ci). Cet intervalle de temps représente donc, dans le pire cas, un gâchis des ressources du réseau dans le but de garantir leurs disponibilité pour la fenêtre suivante. Il est préférable de limiter l'emploi du TAS au trafic critique afin de minimiser ce gâchis. Il n'est donc pas souhaitable de généraliser l'utilisation du TAS à du trafic moins critique, ayant une plus grande marge pour respecter leur contraintes temps réel, et de privilégier l'utilisation d'autres mécanismes d'ordonnancement pour les autres types de trafic.

La figure 7.1 ne présente qu'un exemple simple d'une configuration du TAS. Dans la pratique, il est probable qu'il y ait plusieurs fenêtres protégées et donc plusieurs fenêtres de *guard band* ce

qui accentue encore davantage le phénomène de gâchis des ressources du réseau. En revanche, une répartition de la transmission du trafic critique dans plusieurs fenêtres protégées est une stratégie d'ordonnancement qui permet d'optimiser la latence du reste du trafic. Regrouper la transmission du trafic critique en une seule fenêtre protégée permet de limiter le gâchis dû à la présence de multiple fenêtres de *guard band* mais il est rare que l'ensemble des trames du trafic critique soit prêt à être transmis à un moment donné. Répartir la transmission de ce trafic en plusieurs fenêtres permet donc d'offrir des fenêtres plus nombreuses au reste du trafic.

7.1.2 Les spécificités du CBS

Le CBS a quant à lui pour objectif de répartir l'accès aux ressources du réseau de manière équitable en attribuant une quantité de bande passante suffisante pour respecter les exigences – en termes d'échéances – des flux de données concernés.

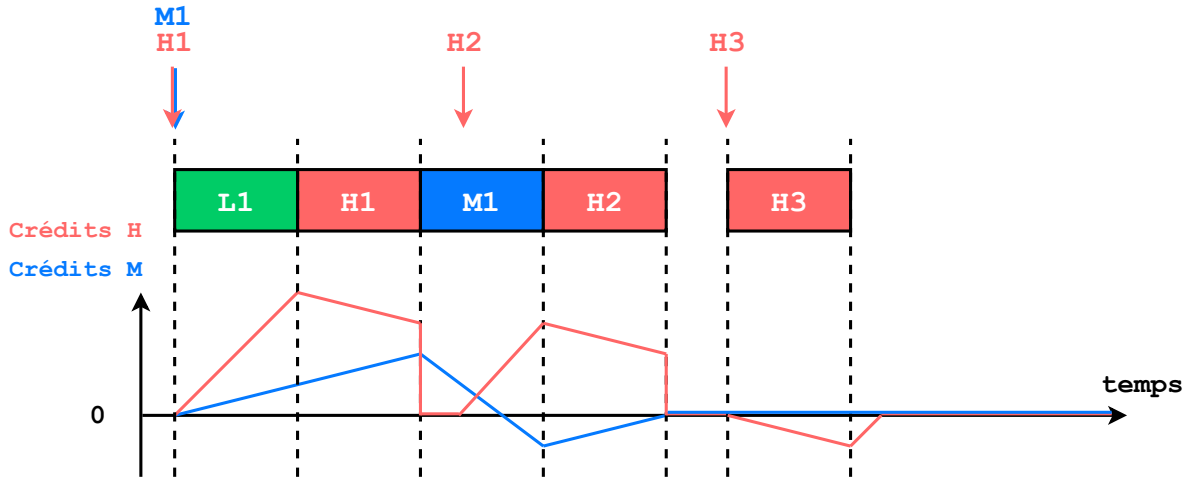


FIGURE 7.2 – Diagramme typique du comportement du CBS.

La figure 7.2 présente le comportement du CBS dans un cas typique de son utilisation, où des flux de données de trois niveaux de priorité (H, M et L) partagent les ressources du réseau pendant une fenêtre libre (non occupée par une *guard band* ou une fenêtre réservée au trafic de type contrôle-commande). Les flux de données des niveaux de priorité H (haute) et M (moyenne) sont gérés par le CBS. Les flèches verticales indiquent le moment d'arrivée de chaque trame, les rectangles indiquent la durée de transmission de chaque trame.

Les courbes présentent l'évolution de la quantité de crédit associée à chaque niveau de priorité géré par le CBS dans le temps. La courbe rouge correspond à la quantité de crédit associée au niveau de priorité des trames rouges, idem pour la courbe bleue. Les trames rouges ont un niveau de priorité plus élevé que les trames bleues qui ont elles-mêmes un niveau de priorité plus élevé que les trames vertes qui elles n'ont pas un niveau de priorité géré par le CBS.

Comme le montre l'évolution des quantités de crédit, il arrive des moments pendant lesquels cette quantité est strictement négative et empêche donc la transmission de trames du niveau de priorité affecté jusqu'au retour de cette quantité à une valeur positive ou nulle. Cette spécificité du comportement du CBS conduit à décourager son utilisation pour l'ordonnancement du trafic le plus critique. En effet, les périodes pendant lesquelles la transmission des trames d'un niveau de priorité donné n'est temporairement plus possible vont augmenter la latence de ces trames.

7.1.3 Conclusion

Un des intérêts centraux de TSN est de regrouper plusieurs types de trafic sur un seul et même réseau. La présence de plusieurs mécanismes d'ordonnancement différents dans TSN est donc essentielle car cela permet de pouvoir s'adapter aux besoins de chaque types de trafic.

Chacun de ces mécanismes d'ordonnancement possède ses propres spécificités et nécessite une méthode pour produire sa configuration. Dans notre cas, la combinaison du TAS et du CBS est particulièrement bien adaptée aux types de trafic que nous rencontrons dans le système que nous concevons mais elle demande également un processus de configuration particulier car elle représente une spécificité en elle-même.

En effet, cette combinaison de deux mécanismes d'ordonnancement crée une interaction entre eux qui va invalider les méthodes de configuration du CBS utilisées lorsqu'il est le seul mécanisme d'ordonnancement présent. Cette interaction est abordée par [MS17] et sera présentée dans la section suivante.

7.2 Les problèmes de configuration du CBS

Des méthodes de calcul de la configuration du CBS censées permettre le respect de leurs exigences de latence de bout en bout existent. En revanche, ces méthodes possèdent deux problèmes :

— les différentes méthodes de calcul présentes dans les standards ne permettent pas d'obtenir un résultat correct dans tous les cas ;

— la présence du TAS n'est pas prise en considération dans le calcul de la configuration du CBS.

7.2.1 Méthodes de calcul de l'état de l'art

Les différents standards de TSN proposent trois méthodes de calcul différentes pour le délai local dans le pire cas pour chaque pont faisant partie du chemin d'un flux de données géré par le CBS. Ces méthodes de calcul du délai utilisent la proportion de bande passante réservée – c'est-à-dire la valeur *idle slope*, qui correspond à la configuration du CBS, comme présenté dans la chapitre 2 – comme variable, ce qui permet donc son calcul en fonction de la contrainte de latence à respecter.

Ces trois méthodes de calcul sont présentes dans les standards [Q18], [BA21] et [Ful09]. Les travaux présentés dans [MVG+23] reviennent en détail sur ces méthodes et prouvent qu'elles sont erronées de façon formelle et par le biais de contre-exemples en simulation. Cet article propose également une nouvelle méthode de calcul de la configuration du CBS plus robuste.

Cette nouvelle méthode de calcul se base néanmoins sur une analyse locale à chaque pont ce qui crée du pessimisme, comme ce sera présenté dans la suite de ce chapitre. De plus, la méthode de calcul proposée est limitée à l'utilisation unique du CBS et ne prend donc pas en considération la présence d'autres mécanismes d'ordonnancement, notamment le TAS.

En dehors des standards, un ensemble de travaux existent sur l'analyse du délai des flux de données gérés par le CBS et donc sur sa configuration. Trois approches différentes ont été particulièrement étudiées : l'approche par *Network Calculus* [DAB14a], l'approche par trajectoire [DAB14b] et plus récemment l'approche par intervalles éligibles [CCBL16b], [CCBL16a] et [CCBL18].

Parmi ces approches, celle par intervalles éligibles est particulièrement intéressante car elle obtient les résultats minimisant le plus le pessimisme, comme présenté dans [CCBL18].

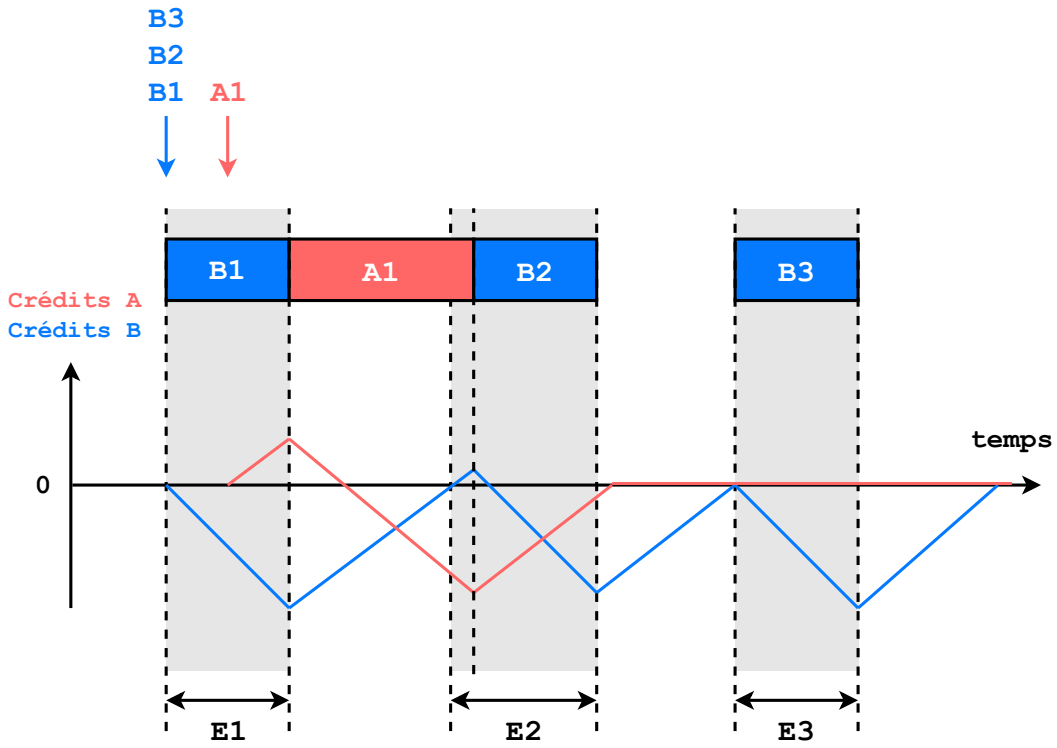


FIGURE 7.3 – Diagramme présentant un exemple de comportement du CBS comportant trois intervalles éligibles.

La figure 7.3 présente un exemple d'intervalles éligibles. Elle reprend le même système de représentation que la figure 7.2.

Un intervalle éligible est un intervalle de temps pendant lequel un flux de données a des trames en attente de transmission et pendant lequel le niveau de priorité qui lui est associé possède une quantité de crédit positive ou nulle. C'est un intervalle de temps pendant lequel les trames peuvent être transmises sauf si le port de sortie est occupé par des trames appartenant à un autre niveau de priorité. Dans le cas présenté ici, trois intervalles éligibles sont présents, identifiés par les intervalles E1, E2 et E3.

La première contribution, publiée dans [CCBL16a], présente l'analyse par intervalles éligibles pour les trames gérées par le CBS avec des interférences de flux de données de priorités inférieure ou supérieure à la trame étudiée. La présente des deux types d'interférences en même temps est mentionnée comme travaux futurs.

Par la suite, dans [CCBL16b], cette approche est étendue pour inclure la présence des deux types d'interférences en même temps. Cette analyse ne repose sur aucune hypothèse concernant le niveau de priorité du trafic causant des interférences, il n'est pas nécessaire de connaître les contraintes qui s'appliquent à ce trafic. L'indépendance vis-à-vis du trafic causant les interférences est permis par le comportement du CBS, qui gère le gain et la dépense des crédit associés aux différents niveaux de priorités.

Dans [CCBL18], cette approche est encore étendue à un cas général permettant de prendre en considération la présence d'un nombre arbitraire de trames de priorité inférieure et supérieure. La prise en compte de toutes les trames de priorité supérieure empêche de maintenir la garantie de minimisation du gâchis des ressources du réseau mais les résultats obtenus demeurent plus

optimisés que ceux qu'il est possible d'obtenir par l'approche par trajectoire. Nous remarquons que le traitement du cas de l'utilisation conjointe du CBS et du TAS est mentionné dans les travaux futurs et que cette analyse reste une analyse locale à chaque pont.

Enfin, dans [CAC+18], les mêmes auteurs proposent deux algorithmes permettant de calculer la valeur minimale de bande passante à réserver (*idle slope*) pour satisfaire les contraintes des flux de données. En revanche, ces algorithmes imposent que la latence de bout en bout soit égale à la contrainte à respecter pour pouvoir automatiquement déduire la valeur de la bande passante à réserver.

Ces travaux permettent de facilement calculer la configuration du CBS dans le cas où il est utilisé seul, en obtenant les valeurs de *idle slope* et *send slope*. Les valeurs ainsi obtenues ont été prouvées comme minimisant le gâchis des ressources du réseau, au moins dans le cas où seule deux niveaux de priorités gérés par le CBS sont utilisés. Cela fait de cette approche un candidat intéressant pour une extension incluant la prise en compte de la présence du TAS.

Un modèle incluant l'utilisation conjointe de TAS et de CBS a été analysé dans [ZPZL18] en utilisant le *Network Calculus* pour calculer la latence dans le pire cas des flux de données gérés par le CBS en présence du TAS et du mécanisme de préemption de trames.

L'utilisation de la préemption de trames permet de réduire la taille des *guard band* qu'il faut utiliser dans le cadre du TAS, ce qui compense l'utilisation supplémentaire de bande passante que ce mécanisme entraîne. En revanche, les résultats de ces travaux contiennent un important pessimisme et sont limités à l'utilisation de seulement deux niveaux de priorités gérés par le CBS.

Dans le rapport technique [ZPZ+18], les mêmes auteurs proposent une amélioration de cette approche levant la restriction du nombre de niveaux de priorités pris en charge et réduisant le pessimisme. Néanmoins, cette approche fait l'hypothèse qu'il est impossible de transmettre de trames durant une fenêtre de *guard band* alors que le standard prévoit qu'il est possible de finir de transmettre des trames pendant cette période tant que leur transmission est terminée avant la fin de la fenêtre de *guard band*. Ce cas de figure se produit lorsque la taille de la fenêtre de *guard band* est fixée à la taille de la plus grande trame d'un niveau de priorité inférieur à ceux gérés par le CBS. Si ces trames d'un niveau de priorité inférieur sont de grande taille, ignorer ce cas de figure conduit à un important pessimisme car l'entièreté de la durée des fenêtres de *guard band* est considérée comme perdue.

7.2.2 Le problème de l'interaction entre TAS et CBS

Comme mentionné précédemment, lorsque le TAS et le CBS sont utilisés de façon conjointe dans un réseau TSN, une interaction entre les deux prend place. Cette interaction peut avoir un impact important sur les flux de données gérés par le CBS.

La figure 7.4 reprend le même système de notation que la figure 7.2. Elle présente l'interaction entre le TAS et le CBS. Le premier diagramme présente le comportement du CBS dans un cas où il est utilisé seul, le deuxième diagramme dans un cas où il est utilisé conjointement au TAS. Dans les deux cas, l'ensemble des paramètres (ordre d'arrivée des trames, taille des trames, configuration du CBS, etc.) sont identiques, la seule différence est la présence du TAS.

On remarque que dans le cas du deuxième diagramme, la présence d'une fenêtre protégée entraîne une augmentation de la latence – c'est-à-dire la différence entre le moment d'arrivée, symbolisé par une flèche verticale, et le moment de transmission, symbolisé par un rectangle – des trames rouges et bleues, appartenant à des flux de données gérés par le CBS. De plus, l'ordre de transmission de ces trames est différent de celui visible dans le cas où le CBS est utilisé seul. La différence est particulièrement importante pour la trame M1, d'un niveau de priorité inférieur

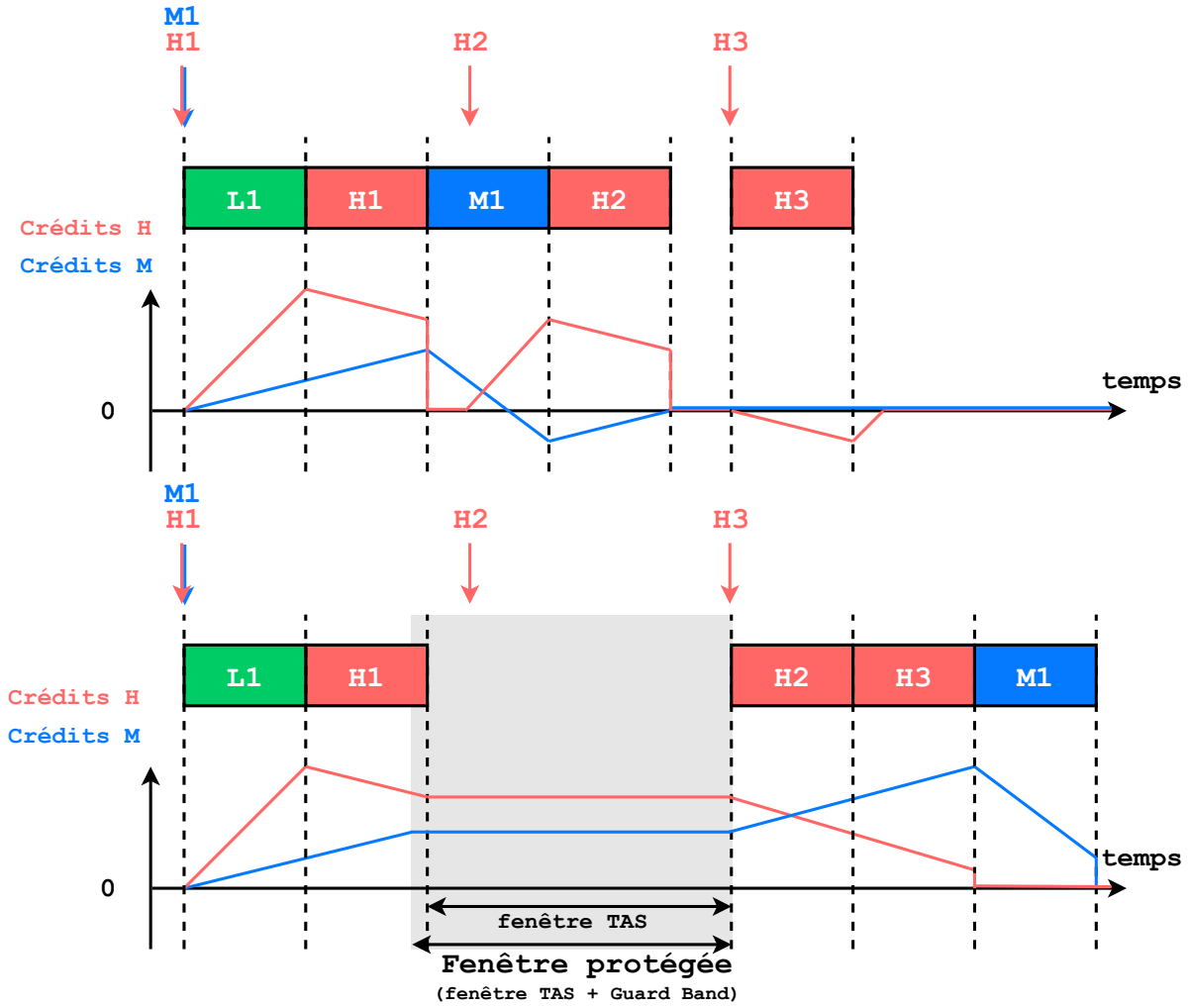


FIGURE 7.4 – Diagramme du comportement du CBS seul et du CBS en présence du TAS dans un cas identique

à celui des trames rouges.

Les standards TSN ne spécifient pas que les valeurs de crédits doivent être gelées pendant les fenêtres de *guard band*. En revanche, il a été démontré dans [BD19] que si ces valeurs ne sont pas gelées pendant les fenêtres de *guard band*, et qu'elles peuvent donc augmenter, cela peut mener à une accumulation disproportionnée de crédits ce qui conduit à un phénomène de famine pour les flux de données de niveaux de priorité inférieurs.

Dans [ZPZL18], les auteurs proposent l'option de geler l'évolution des valeurs du crédit pendant les fenêtres de *guard band*. Une évaluation des pires temps de réponses des flux de données gérés par le CBS avec et sans gel des valeurs de crédits est proposée dans [ZPZ+21].

Nous faisons donc l'hypothèse que lorsque le niveau de priorité d'un flux géré par le CBS n'est pas autorisé à transmettre, parce que le fenêtre du TAS en cours ne le permet pas, la valeur de crédit qui lui est associé est gelée et ne peut donc pas augmenter comme c'est le cas habituellement (c'est le cas pour les deux valeurs de crédits de la figure 7.4). En revanche, les trames d'un flux de données géré par le CBS peuvent terminer leur transmission pendant une fenêtre de *guard band* si elle a débuté avant le début de celle-ci. Dans ce cas, la quantité de crédit

associé au niveau de priorité des trames décroit normalement, comme le montre l'évolution de la valeur de crédits représentée par la courbe rouge lors de la transmission de la trame *H1* sur la figure 7.4.

La transmission de la trame bleue ne peut pas être bloquée indéfiniment par la transmission de trames appartenant à des flux de données d'un niveau de priorité supérieur gérés par le CBS car son fonctionnement entraîne obligatoirement un passage de la quantité de crédit à une valeur négative. Néanmoins, la présence du TAS peut, comme c'est illustré ici, grandement nuire à la latence des trames des flux de données gérés par le CBS.

7.2.3 Conclusion

Des méthodes de calcul de la latence de bout en bout des flux de données gérés par le CBS existent. Ces méthodes permettent le calcul de la configuration du CBS permettant de respecter les contraintes de latence.

Néanmoins, les méthodes proposées dans les standards TSN sont erronées, comme cela a été prouvé dans [MVG+23]. De plus, les méthodes proposées dans l'état de l'art, bien qu'utilisant différentes approches, ne prennent pas en considération la présence du TAS aux côtés du CBS et ne permettent donc pas de produire une configuration du CBS correcte dans ce cas.

Les rares approches prenant la présence du TAS en compte proposent des résultats particulièrement pessimistes du fait de la non prise en compte des transmission qui peuvent avoir lieu pendant les fenêtres de *guard band*.

Nous avons démontré l'impact que peut avoir la présence de fenêtres protégées sur les délais de transmission des flux de données gérés par le CBS. Il est clair qu'une approche de calcul de la configuration du CBS prenant en compte la présence du TAS est nécessaire et nous proposons dans ce chapitre de nous baser sur l'approche par intervalles éligibles pour proposer une telle méthode.

7.3 Calcul de la configuration du CBS pour un pont utilisant aussi le TAS

Dans cette section, nous présentons une méthode de calcul de la configuration du CBS au niveau d'un port de sortie d'un pont du réseau lorsque le CBS est utilisé conjointement au TAS. L'approche que nous présentons est basée sur l'approche par intervalles éligibles définies dans [CCBL16b] et [CCBL16a] et l'enrichit pour prendre en considération le blocage des transmissions de certains niveaux de priorités causé par le TAS. Ces travaux s'appuient sur les résultats présentés dans [MS17] en les étendant pour leur ajouter les règles explicites permettant de calculer la proportion de bande passante à réserver – c'est-à-dire le paramètre *idle slope*, représentant la configuration du CBS – pour un port d'un switch de façon à respecter les exigences des flux de données gérés par le CBS.

Le type de réseaux TSN que nous voulons pouvoir configurer ont donc la possibilité de transmettre plusieurs types de trafic différents, chacun avec ses propres contraintes : le trafic critique (TC), typiquement des flux de données de contrôle-commande, le trafic audio (A), le trafic vidéo (B), A et B sont le trafic géré par le CBS, le trafic dit « meilleur effort » (BE), pour lequel les mécanismes d'ordonnancement n'ont pas de gestion particulière à faire. Ce dernier type de trafic est celui qui n'a aucune contrainte et qui peut donc avoir accès aux ressources du réseaux restantes après la transmission des autres types de trafic.

Nous commençons par présenter l'approche par intervalles éligibles dans un contexte où seul le CBS est utilisé, puis nous y ajoutons la prise en considération de la présence du TAS dans le cas où il ne peut y avoir qu'une unique fenêtre protégée par cycle TAS, et donc une unique *guard band*, puis nous généralisons ce résultat pour le cas général, dans lequel il peut y avoir un nombre arbitraire de fenêtres protégées, et donc de fenêtres de *guard band*.

La table 7.1 regroupe les symboles utilisés dans la notation présente dans ce chapitre.

Notation	Définition
TC, A, B, BE	Classes de trafic correspondant au trafic critique, audio, vidéo et « meilleur effort »
L_{TAS}	Durée du cycle TAS
L_{TC}	Durée d'une fenêtre temporelle pour le trafic critique
L_{GB}	Durée d'une fenêtre de *guard band*
L_{PW}	Durée totale d'une fenêtre protégée, $L_{PW} = L_{TC} + L_{GB}$
f_i	Un flux de données f de classe i
$f_j \in eqp(i)$	Les autres flux de données de la classe i
C_i	Temps de transmission d'une trame d'un flux de données de la classe i
T_i	Période de transmission ou intervalle d'interarrivée d'un flux de données de la classe i
D_i	Échéance d'un flux de données de la classe i
BW	Bande passante disponible à un port d'un pont
α_i^+ et α_i^-	Les paramètres *idle slope* et *send slope* (la configuration du CBS) de la classe i
α_H^+ et α_H^-	Les paramètres *idle slope* et *send slope* d'une classe d'un niveau de priorité supérieur à celui de la classe i
α_L^+ et α_L^-	Les paramètres *idle slope* et *send slope* d'une classe d'un niveau de priorité inférieur à celui de la classe i
$R_{FIFO}(f_i)$	Délai dans le pire cas du flux de données f_i dans une file (FIFO)
$R_{CBS}(f_i)$	Délai dans le pire cas du flux de données f_i dû à l'utilisation du CBS
$R_{TSN}(f_i)$	Délai total dans le pire cas du flux de données f_i
BR_i	Ratio de réservation de bande passante de la classe i
U_i, U_H, U_L	Charge des classes i, d'un niveau de priorité supérieur et d'un niveau de priorité inférieur, respectivement

TABLE 7.1 – Table des symboles utilisés dans la notation de ce chapitre.

7.3.1 L'approche par intervalles éligibles dans le cas de l'utilisation du CBS seul

Nous présentons ici l'approche par intervalles éligibles de l'analyse de la latence dans le pire cas d'un flux de données géré uniquement par le CBS, présentée dans [CCBL16b]. Cette analyse calcule la latence dans le pire cas d'une trame d'un flux f_i en prenant en compte le temps de blocage dans le pire cas par des trames d'un niveau de priorité supérieur (C_H^{max}) ainsi que celui causé par les trames d'un niveau de priorité inférieur (C_L^{max}). Dans la suite de ce chapitre, nous utiliserons la lettre H pour indiquer la classe de trafic d'un niveau de priorité supérieur à celui du flux de données f_i, la lettre M pour la classe de trafic de f_i (la classe de la trame étudiée) et

la lettre L pour indiquer la classe de trafic d'un niveau de priorité inférieur à celui de f_i.

L'équation (venant de [CCBL16b]) permettant de calculer cette latence est la suivante :

$$R_{CBS}(f_i) = R_{FIFO}(f_i) + C_L^{max} \times (1 + \frac{\alpha_H^+}{\alpha_H^-}) + C_H^{max} \tag{7.1}$$

Dans l'équation 7.1, $R_{FIFO}(f_i)$ correspond au délai causé par la mise en file d'attente des trames d'une même classe de trafic au niveau du port par lequel la trame étudiée va être transmise. Cette valeur se calcule de la façon suivante (venant également de [CCBL16b]) :

$$R_{FIFO}(f_i) = C_i + \sum_{f_j \in eqp(i)} C_j \times (1 + \frac{\alpha_i^-}{\alpha_i^+}) \tag{7.2}$$

Dans l'équation 7.1, C_L^{max} correspond au temps de transmission maximum d'une trame d'un flux de données de la classe L. Cette valeur représente le temps de blocage causé par une trame d'un niveau de priorité inférieur ayant commencé sa transmission avant que la trame étudiée ne soit disponible pour sa transmission. Il est important de noter que nous faisons l'hypothèse que le mécanisme de préemption de trames n'est pas actif dans le cadre ce notre analyse. La valeur C_H^{max} représente le temps de blocage maximum d'une trame d'un flux de données de la classe H.

Dans l'équation 7.2, $eqp(i)$ représente l'ensemble des flux de données de même niveau de priorité que la trame étudiée. Les valeurs α_i^+ et α_i^- correspondent aux paramètres *idle slope* et *send slope* de la classe de la trame étudiée, avec $\alpha_i^+ + \alpha_i^- = BW$. Ce sont des proportions de bande passante. L'utilisation maximale des ressources du réseau – c'est-à-dire de la bande passante ici – de la classe i est donc $\frac{\alpha_i^+}{BW}$, cela arrive lorsque seule des trames de la classe i sont transmises, sans interruption.

Il est important de noter que pour que les équations 7.1 et 7.2 soient valides, la latence de transmission de la classe i doit être bornée, ce qui implique de respecter la condition d'utilisation des ressources du réseau donnée par l'équation 7.3 :

$$\sum_{eqp(i)} \frac{C_i}{T_i} \leq \frac{\alpha_i^+}{BW} \tag{7.3}$$

Les travaux présentés dans [CCBL16b] prouvent que l'équation 7.1 permet d'obtenir un résultat moins pessimiste que celui obtenu par les autres approches.

La figure 7.5 présente un diagramme du pire cas pouvant se produire pour la latence de la trame étudiée (la trame bleue). Cette trame est d'abord retardée par le délai dû à la file d'attente, ce délai correspond aux trames de même priorité (classe M), puis par une trame de classe L, puis par plusieurs trames de classes H. Le nombre de trames de classe H pouvant augmenter la latence de la trame étudiée dépend de la quantité de crédit accumulée pendant la transmission de la trame de classe L (égal à la valeur du paramètre *idle slope* de la classe H multiplié par C_L^{max}). Ce pire cas mène à l'équation 7.1.

Une fois le temps de transmission d'une trame appartenant à un flux de données géré uniquement pas le CBS calculé, nous pouvons inclure le blocage supplémentaire dû à l'utilisation du TAS et au blocage supplémentaire causé par la présence de fenêtres protégées qui empêchent la transmission de ces trames.

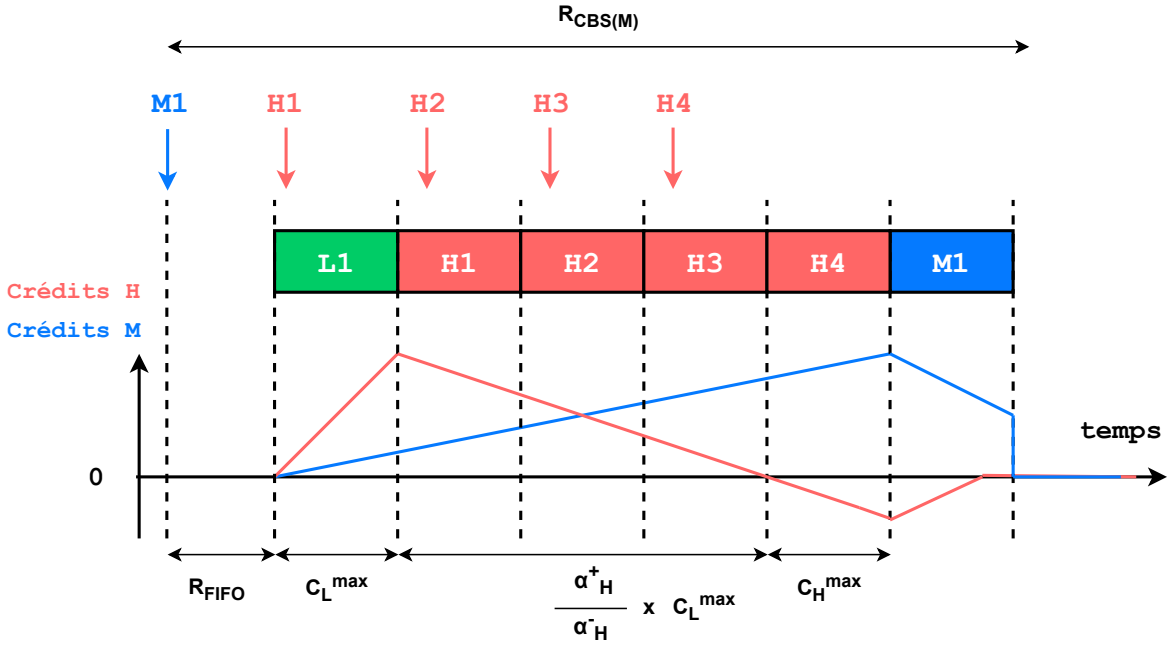


FIGURE 7.5 – Illustration du pire cas pour la latence de la trame étudiée sans présence du TAS.

7.3.2 Calcul de la configuration du CBS dans le cas d'une unique fenêtre protégée

Dans cette sous-section, nous présentons la borne sur la latence de la trame étudiée dans le cas où le CBS est utilisé conjointement au TAS et le moyen de calculer la configuration du CBS à utiliser pour respecter les contraintes de la trame étudiée, dans le cas où il ne peut y avoir qu'une unique fenêtre protégée par cycle TAS. La présence d'une unique fenêtre protégée peut correspondre à une stratégie de configuration du TAS qui consiste à minimiser la présence de fenêtre de *guard band* en regroupant toutes les transmissions du trafic critique en une seule fenêtre. Nous faisons néanmoins l'hypothèse que l'échéance de la trame étudiée est inférieure à la durée d'un cycle TAS, cette hypothèse est généralement respectée car la durée du cycle TAS est basé sur l'hyperpériode de transmission des flux de données appartenant au trafic critique.

Analyse de la latence

Nous appelons L_{TC} la durée de la fenêtre temporelle dédiée à la transmission du trafic critique, c'est-à-dire la durée pendant laquelle les ressources du réseau ne sont accessible que par ce type de trafic. De la même façon, nous appelons L_{GB} la durée de la fenêtre de *guard band* qui précède la fenêtre temporelle dédiée au trafic critique, c'est-à-dire la durée pendant laquelle aucune nouvelle transmission ne peut commencer mais qui permet néanmoins à une transmission déjà en cours de se terminer. Enfin, L_{PW} désigne la somme de ces deux valeurs, qui correspond à la durée totale de la fenêtre protégée.

La quantité de crédits associée aux classes de trafic gérées par le CBS ne peut augmenter lorsque leur transmission est rendue impossible par une fenêtre protégée, conformément à notre hypothèse, également faite par [MS17], qui est celle du gel des valeurs de crédits présentée dans [ZPZ+21]. Cette fenêtre a pour effet de geler l'augmentation de cette quantité de crédits, qui pourra reprendre une fois qu'elle sera terminée dans l'état atteint avant qu'elle ne commence. Il est important de noter que la quantité de crédits peut diminuer pendant une fenêtre de *guard*

band si une trame termine sa transmission pendant cette dernière, ce qui ne s'oppose pas aux analyses présentées dans [ZPZ+21] et [BD19].

Définition 7.3.1. (Ratio de réservation). Le ratio de réservation de la classe i, BR_i, est la proportion de bande passante accordée à la classe i par le biais des paramètres *idle slope* et *send slope* après lui avoir soustrait la proportion de bande passante monopolisée par la fenêtre protégée :

$$BR_i = \frac{\alpha_i^+}{\alpha_i^+ + \alpha_i^-} \times (1 - \frac{L_{GB} + L_{TC}}{L_{TAS}}) \tag{7.4}$$

avec L_{TAS} la durée d'un cycle TAS.

Définition 7.3.2. (Condition de faisabilité). Afin de garantir que le temps de transmission dans le pire cas d'une trame appartenant à une classe de trafic gérée par le CBS n'est pas infini, une condition nécessaire est que la charge de la classe i est inférieure à son ratio de réservation :

$$U_i = \sum_{eqp(i)} \frac{C_i}{T_i} \leq BR_i \tag{7.5}$$

Théorème 7.3.1. Si la condition de faisabilité données par l'équation 7.5 est respectée, alors la latence dans le pire cas d'un flux de données f_i appartenant à une classe de trafic gérée par le CBS dans un pont peut se calculer de la façon suivante :

$$R_{TSN}(f_i) = R_{CBS}(f_i) + L_{PW} \tag{7.6}$$

Preuve du théorème 7.3.1. Un effet de la présence du TAS qui participe grandement à la définition du temps de transmission dans le pire cas d'une trame appartenant à une classe de trafic gérée par le CBS est présenté dans la figure 7.4, qui montre un changement de l'ordre de transmission des trames causé par la présence d'une fenêtre protégée.

Le nombre de trames de classe H pouvant bloquer la trame de classe M – la trame étudiée – est limité par les paramètres *idle slope* et *send slope* de la classe H car la quantité de crédits associée à cette classe ne peut augmenter pendant la fenêtre protégée. De ce fait, peu importe le nombre de trames de classe H prêtes à être transmises à la fin de la fenêtre protégée, seul un sous-ensemble de ces trames pourront effectivement causer un blocage avant que la quantité de crédits atteigne une valeur strictement négative. Cette limite au blocage par des trames d'un niveau de priorité supérieur est visible sur la figure 7.5 dans laquelle il n'y a pas de fenêtre protégée.

La quantité de crédits associée à la classe H peut atteindre une valeur maximum de $\alpha_H^+ \times C_L^{max}$ avant de décroitre à un taux α_H^- (correspondant au paramètre *send slope*) et met $\frac{\alpha_H^+ \times C_L^{max}}{\alpha_H^-}$ unités de temps à atteindre une valeur nulle, à ce point, la classe H peut encore transmettre exactement une trame, avec un temps de transmission C_H^{max}.

Ce scénario conduit à la latence dans le pire cas de la trame de classe M dans le cas où seul le CBS est utilisé, donné par l'équation 7.1. Puisque la quantité de crédit associée à la classe H ne peut augmenter pendant la fenêtre protégé, cette borne supérieure sur le blocage causé par des trames d'un niveau de priorité supérieur reste vraie pour le cas de l'utilisation conjointe du TAS et du CBS, ce qui abouti à la latence dans le pire cas d'une trame appartenant à une classe de trafic gérée par le CBS, donnée par l'équation 7.6. □

Les équations 7.4, 7.5 et 7.6 fournissent un cadre d'analyse suffisant pour calculer la latence des trames appartenant aux classes de trafic gérées par le CBS lorsque le TAS est également utilisé et permet donc aussi de vérifier l'ordonnancement du réseau en comparant le résultat de ce calcul avec les échéances des flux de données.

Calcul de la configuration du CBS

L'algorithme de réservation de la proportion minimale de bande passante proposé dans [CAC+18] peut être étendu au contexte de l'utilisation conjointe du TAS et du CBS entraînant un blocage dû à la fenêtre protégée. Cette analyses reste donc également locale à un pont. Nous considérons les paramètres α_H^+ et α_H^- (respectivement *idle slope* et *send slope*) comme associés au trafic audio (A) et les paramètres α_M^+ et α_M^- comme associés au trafic vidéo (B). Les valeurs des paramètres α_H^+ et α_M^+ sont données par les équations 7.7 et 7.9.

La valeur minimale de α_H^+ est donnée par l'équation suivante :

$$\alpha_H^+ \geq max(\frac{U_H}{1-\frac{L_{PW}}{L_{TAS}}}, \frac{\sum_{f_j \in H, j \neq i} C_j}{D_i - C_i - C_{M,L}^{max} - L_{PW}}) \times BW \tag{7.7}$$

L'équation 7.7 donne une borne inférieure sur la proportion de bande passante à attribuer au niveau de priorité associé à la classe H permettant de respecter la contrainte de latence du flux de données f_i. Cette valeur peut être égale à $\frac{U_H}{1-\frac{L_{PW}}{L_{TAS}}} \times BW$ (obtenue à partir de l'équation 7.4) si la contrainte de faisabilité est dominante, i.e. l'échéance a une large marge. Elle est égale à $\frac{\sum_{f_j \in H} C_j}{D_i - C_i - C_{M,L}^{max} - L_{PW}} \times BW$ (obtenue à partir de l'équation 7.6 en posant $R_{TSN}(f_i) = D_i$) si la contraite de l'échéance est dominante, i.e. si la marge donnée par l'échéance est si faible qu'il est nécessaire de réserver davantage de bande passante.

Nous notons une fois de plus que ce résultat n'est valable que si $D_i \leq L_{TAS}$. Si cette contrainte n'est pas respectée, il est possible de poser une échéance virtuelle : $D_i' = min(D_i, L_{TAS})$.

La valeur de α_H^+ doit également respecter la borne supérieure donnée par l'équation suivante :

$$\alpha_H^+ \leq (1 - \frac{L_{PW}}{L_{TAS}}) \times BW \tag{7.8}$$

L'équation 7.8 donne la borne supérieure sur la bande passante que cette classe de trafic peut utiliser. Elle correspond à la bande passante disponible à laquelle on soustrait la proportion utilisée par la fenêtre protégée.

De la même façon, la valeur de α_M^+ peut être obtenue par l'équation suivante :

$$\alpha_M^+ \geq max(\frac{U_M}{1-\frac{L_{PW}}{L_{TAS}}}, \frac{\sum_{f_j \in M, j \neq i} C_j}{D_i - C_i - C_L^{max}(1+\frac{\alpha_H^+}{\alpha_H^-}) - C_H^{max} - L_{PW}}) \times BW \tag{7.9}$$

L'équation 7.9 donne la proportion minimale de bande passante à attribuer au niveau de priorité associé à la classe M permettant de respecter la contrainte de latence du flux de données f_i. La sélection du terme dominant de la fonction $max()$ est faite de la même façon que pour l'équation 7.7.

$$\alpha_M^+ \leq (1 - \frac{L_{PW}}{L_{TAS}}) \times BW - \alpha_H^+ \tag{7.10}$$

L'équation 7.10 donne la borne supérieure de la bande passante que cette classe de trafic peut utiliser. Elle correspond à la bande passante disponible à laquelle on soustrait la proportion utilisée par le fenêtre protégée et celle utilisée par la classe H.

Les équations 7.9 et 7.10 peuvent être étendues au cas général dans lequel il peut y avoir plus d'une classe de trafic d'un niveau de priorité supérieur à celui de la trame étudiée en soustrayant à la bande passante totale disponible au niveau du port la proportion de bande passante réservée pour toutes les classes d'un niveau de priorité supérieur.

7.3.3 Généralisation à un nombre arbitraire de fenêtres protégées

Le cas général dans lequel il peut y avoir un nombre arbitraire de fenêtres protégées est plus complexe. Nous appelons L_{TC}^k la durée de la k-ème fenêtre temporelle dédiée au trafic critique, dans l'ensemble des n fenêtres de ce type présentes dans un cycle TAS. Chacune de ces fenêtres est précédée par une fenêtre de *guard band* d'une durée L_{GB}.

Une borne supérieure sur la latence dans le pire cas d'une trame appartenant à une classe de trafic gérée par le CBS lorsque le TAS est également présent est obtenue en généralisant l'équation 7.6 :

$$R_{TSN}(f_i) = R_{CBS}(f_i) + \sum_{k=1}^{n} L_{TC}^k + n \times L_{GB} \tag{7.11}$$

Cette équation doit également respecter la condition de faisabilité généralisée :

$$\sum_{eqp(i)} \frac{C_i}{T_i} \leq BR_i^n \tag{7.12}$$

avec

$$BR_i^n = \frac{\alpha_i^+}{BW} \times (1 - \frac{n \times L_{GB} + \sum_{k=1}^{n} L_{TC}^k}{L_{TAS}}) \tag{7.13}$$

Afin de pouvoir effectuer ces calculs, il est nécessaire de connaître la durée de chaque fenêtre temporelle dédiée au trafic critique dans le pont.

Ce résultat est toutefois pessimiste car nous faisons l'hypothèse que toutes les fenêtres protégées sont placées les unes à la suite des autres. Cela cause du pessimisme car dans la pratique, il est probable qu'il existe un écart entre ces fenêtres permettant de transmettre des trames appartenant aux flux de données des classes de trafic gérées par le CBS. Cette analyse peut donc encore être affinée car tous les flux de données ne seront pas bloqués par toutes les fenêtres protégées. En effet, plus le niveau de priorité des flux de données appartenant aux classes de trafic gérées par le CBS est élevée, moins leurs trames seront bloquées par les fenêtres protégées.

Le cadre d'analyse donné par les équations 7.11, 7.12 et 7.13 est une généralisation directe du cas dans lequel il ne peut y avoir qu'une unique fenêtre protégée et se justifie donc de la même façon, avec toutefois un pessimisme plus important. Pour s'assurer que les latences soient finies, il faut s'assurer que les équations 7.7 et 7.9 sont satisfaites en considérant que $L_{PW} = n \times L_{GB} + \sum_{k=1}^{n} L_{TC}^k$.

7.4 Discussion et conclusion

Dans ce chapitre, nous avons présenté l'intérêt et la problématique de l'utilisation conjointe de deux mécanismes d'ordonnancement de TSN : le TAS et le CBS. La présence simultanée de

plusieurs types de trafic sur un même réseau TSN rend l'emploi conjoint de différents mécanismes d'ordonnancement intéressant afin de pouvoir tirer parti de leurs spécificités respectives et de s'adapter aux exigences des différents types de trafic. Néanmoins, l'utilisation combinée de ces deux mécanismes d'ordonnancement entraîne une interaction entre eux qui empêche l'utilisation des méthodes existantes pour le calcul de la configuration du CBS.

En effet, la présence des fenêtres protégées définies par le TAS crée un blocage nuisant à la transmission des trames appartenant aux flux de données gérés par le CBS. Il est donc nécessaire de proposer une méthode de calcul de cette configuration prenant en considération la présence du TAS.

Nous proposons une telle approche en nous basant sur une approche déjà existante pour le calcul de la configuration du TAS, la méthode par intervalles éligibles présentée dans [CCBL16b]. Nous nous appuyons sur les travaux publiés dans [MS17], que nous étendons en proposant des formules permettant de calculer une borne inférieure que la proportion de bande passante à allouer aux classes de trafic gérées par le CBS, c'est-à-dire la configuration du CBS.

Nous proposons dans un premier temps une approche de calcul pour le cas dans lequel il y a une unique fenêtre protégée puis nous généralisons ce résultat à un nombre arbitraire de fenêtres protégées. Ces cas correspondent à deux stratégies de configuration du TAS différentes parmi celles dont nous avons connaissance :

— Une unique fenêtre protégée : cette stratégie permet de minimiser le nombre de fenêtres de *guard band* et donc le gâchis potentiel de ressources réseau. En revanche, cette stratégie entraîne une augmentation de la latence des flux de données appartenant à la classe de trafic critique car certaines de leurs trames devront attendre le début de la fenêtre protégée après avoir été reçue par le pont.

— Une répartition homogène des fenêtres protégées : cette stratégie permet de prévoir plusieurs fenêtres protégées espacées dans le cycle TAS par une durée constante. Cela permet de minimiser l'impact du trafic appartenant à la classe critique sur le trafic appartenant aux classes gérées par le CBS en laissant des fenêtres libres entre chaque fenêtre protégée. Pour la même raison qu'avec la stratégie précédente, cela peut augmenter la latence des flux de données de la classe critique.

— Un nombre arbitraire de fenêtres protégées : cette stratégie ne fait aucune hypothèse sur les fenêtres protégées, il peut y en avoir un nombre arbitraire et elles peuvent s'enchaîner de façon plus ou moins espacée. Cela permet de minimiser la latence du trafic critique en ajustant le positionnement dans le cycle TAS et la durée des fenêtres protégées en fonction des besoins de ce trafic. En revanche, cette stratégie peut avoir un impact sur le trafic géré par le CBS en laissant peu de temps entre les fenêtres protégées pour la transmission d'autres types de trafic.

C'est cette dernière stratégie que nous considérons pour la généralisation du calcul de configuration du CBS. Néanmoins, dans notre approche, nous faisons l'hypothèse que les fenêtres protégées s'enchaînent les unes après les autres sans interruption. Une façon de lever cette hypothèse serait de ne plus considérer que toutes les fenêtres protégées bloquent le trafic géré par le CBS puisqu'en pratique ces fenêtres seront espacées, par des fenêtres libres potentiellement d'une très courte durée mais permettant tout de même la transmission d'autres types de trafic. La partie du trafic géré par le CBS ayant le niveau de priorité le plus élevé pourra donc être transmise durant ces fenêtres libres et ne seront pas bloquées par l'intégralité des fenêtres protégées.

Nous faisons également l'hypothèse que le mécanisme de préemption de trames n'est pas utilisé. Prendre en considération la présence de ce mécanisme permettrait de réduire le temps de

blocage causé par des trames d'un niveau de priorité inférieur à celui de la trame étudiée (C_L^{max}). En revanche, de mécanisme entraîne une augmentation de la bande passante utilisée. En effet, la préemption de la transmission d'une trame par une autre d'un niveau de priorité supérieur entraîne une reprise de la transmission et donc une multiplication des en-têtes des trames causant une augmentation de l'utilisation de la bande passante. Afin de lever cette hypothèse, il semble nécessaire de pouvoir quantifier le nombre d'interruptions pour pouvoir calculer le volume de données que représente les en-têtes supplémentaires.

Enfin, la plus grande limite à notre approche de calcul de la configuration est le fait que c'est une approche locale à un port de sortie d'un pont TSN. Cela implique que notre approche considère le pire cas possible pour ce port afin de calculer une configuration du CBS permettant de respecter les exigences temps réel – en termes de latence – des flux de données. Pour calculer la configuration du CBS à l'échelle du réseau entier, c'est-à-dire au niveau de chaque port de sortie par lequel est transmis au moins un flux de données appartenant à une classe de trafic gérée par le CBS, il est nécessaire de répéter l'application de l'approche afin de répartir une échéance globale en une somme de délais locaux.

Cette méthode aura pour résultat une configuration du CBS pessimiste car le pire cas possible pour un port de sortie sera répété à chaque saut le long du chemin du flux de données, ce qui n'est pas possible en pratique grâce au phénomène de sérialisation des trames.

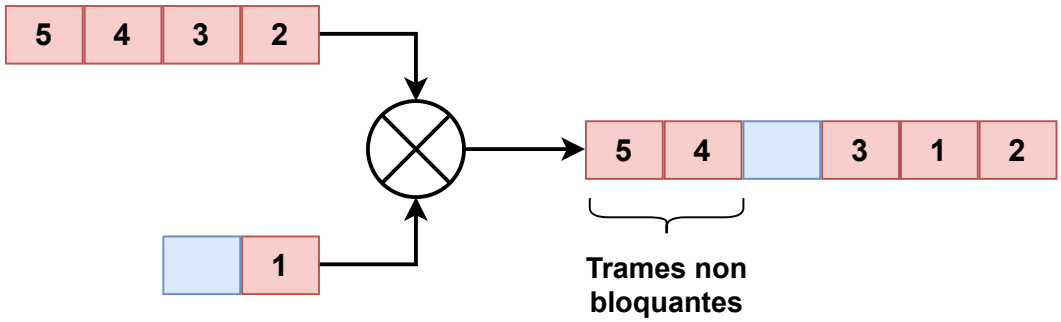


FIGURE 7.6 – Illustration du phénomène de sérialisation des trames.

La figure 7.6 présente une illustration du phénomène de sérialisation de trames. Un pont reçoit deux flux de données, le premier composé de deux trames dont la trame étudiée en bleu, le second composé de quatre trames. Les deux flux sont mis en concurrence au niveau du port de sortie du flux qui transmet les trames les unes après les autres. La trame étudiée est bloquée par la première trame du flux concurrent, qui bloque également la première trame du flux de données principal (numérotée 1), ainsi que par la deuxième trame du flux concurrent. Une fois transmises, l'ensemble de ces trames sont mises bout à bout et les deux dernières trames du flux de données concurrent ne pourront plus bloquer la trame étudiée. La prise en compte de ce phénomène permettrait donc de réduire la valeur du blocage causé par la mise en file d'attente (FIFO).

Une piste que nous envisageons dans le but d'améliorer notre approche de calcul de la configuration du CBS en réduisant son pessimisme est de la combiner avec l'approche par agrégation de flux présentée dans le chapitre 5 de [Doc21]. Une autre piste intéressante serait la combinaison de l'approche par intervalles éligibles avec une approche par *network calculus* car cette dernière permet de mieux prendre en considération le phénomène de sérialisation de trames sur un chemin comportant plusieurs sauts. Une telle approche est présentée dans [ZPZ+21], qui produit une borne de délai de bout en bout plus serrée que celle obtenue avec l'approche que nous proposons.

Enfin, l'approche *Forward end-to-end delay Analysis* (FA) a été appliquée à l'analyse de CBS sans présence du TAS [BBRR18]. Son extension dans le cas de l'utilisation conjointe du TAS et du CBS est une autre piste intéressante.

Les travaux futurs devront donc se concentrer sur la réduction du pessimisme de cette approche en la faisant évoluer d'une approche locale à une approche globale, ne considérant plus un pire cas qui ne peut se produire. La prise en considération de la présence du mécanisme de préemption de trames et la levée de l'hypothèse sur l'enchaînement ininterrompu des fenêtres protégées dans le cas général comportent également un intérêt.

Afin de compléter notre approche de de conception et de configuration de réseaux TSN, il est maintenant nécessaire de pouvoir synthétiser la configuration du TAS et de produire des modèles de simulation et d'analyse pour différents outils permettant de vérifier que la configuration du réseau produite permet bien de respecter l'ensemble des exigences du système.

Chapitre 8

Génération de configuration

Ce chapitre présente notre outil de génération automatique de configuration et de modèles de simulation et d'analyse pour des réseaux TSN : MoBACT (*Model-Based Automatic Configurator for TSN*). Cet outil est la dernière pièce de notre approche, il permet de regrouper les contributions présentées dans les chapitres précédents et il permet leur exploitation.

MoBACT prend comme données d'entrée les modèles présentés dans le chapitre 5 qui décrivent la topologie du réseau, l'ensemble des flux de données qui circuleront sur celui-ci ainsi que les exigences que le système doit être capable de respecter tout au long de son fonctionnement. Lors de son exécution, MoBACT commence donc par parcourir ces modèles afin de former sa représentation interne du problème à résoudre.

Ensuite, si l'option adéquate lui est passée, MoBACT peut faire appel à un outil externe, appelé TSNsched [SSN19], dont le rôle est la synthèse de la partie de la configuration du TAS concernant les fenêtres protégées pour le trafic de la classe critique, qu'il faudra ensuite compléter avec les fenêtres de *guard band* et les fenêtres libres pour les autres types de trafic. Une fois cette synthèse effectuée, il est possible pour l'utilisateur de faire les calculs de la configuration du CBS (c'est-à-dire des valeurs des paramètres *idle slope*) à partir des résultats obtenus, comme présenté dans le chapitre 7.

Enfin, la dernière étape de l'exécution de MoBACT est la production des modèles de simulation et d'analyse. Ces modèles peuvent être générés pour plusieurs cibles différentes : NeSTiNg, RTaW-Pegase et Mininet [20]. En plus de ces modèles, MoBACT permet également de générer une documentation complète du réseau, de ses caractéristiques et de sa configuration sous la forme d'un ensemble de pages HTML qu'il est possible de parcourir facilement avec un navigateur internet. Un graphe de la topologie de réseau est également produit dans un format exploitable par Graphviz. Des fichiers de configuration utilisables par des équipements réseau sont produits, permettant de facilement déployer la configuration du TAS et du CBS sur les ponts utilisés dans le système réel.

8.1 Problématique de la génération de configuration

La conception de réseaux TSN nécessite l'utilisation d'outil (simulateurs réseau et outil de calcul de latence dans le pire cas) afin de vérifier que la configuration qui sera déployée sur le matériel est valide, c'est-à-dire qu'elle permet le respect de toutes les exigences temps réel du système.

20. Mininet ne supporte pas encore TSN mais est utilisé pour rapidement créer un prototype virtuel du réseau sur lequel les applications peuvent être déployées.

Le caractère récent de TSN et ses nombreuses fonctionnalités font que ces outils ne supportent généralement pas l'ensemble des standards et qu'il est donc nécessaire d'en utiliser plusieurs. De plus, tous ces outils n'ont pas les mêmes capacités de simulation et d'analyse et ils ne sont pas tous aussi accessibles (certains sont des outils libres et gratuits, d'autres sont des produits commerciaux.) Enfin, l'utilisation de plusieurs de ces outils permet la comparaison des résultats. Afin d'illustrer notre approche, nous avons sélectionné trois outils de conception différents : Mininet, NeSTiNg et RTaW-Pegase.

Des travaux permettant la génération automatique de configuration existent déjà, comme présenté dans [HBA+21]. Ces travaux permettent la génération automatique de modèles de simulation pour NeSTiNg à l'aide d'une extension pour OMNeT++. Il n'y a pas de définition de ressources permettant la modélisation de réseaux TSN dans ces travaux, seul le format utilisé par OMNeT++ est utilisé. L'intégration à OMNeT++ ne permet pas la génération de modèles de simulation pour d'autres outils. De plus, cette approche ne contient pas de moyen de modéliser automatiquement les flux de données et elle ne permet donc pas d'assurer la cohérence entre le comportement de l'application réelle et le modèle de simulation produit.

8.1.1 Outils de conception

Mininet

Le premier outil que nous avons sélectionné est Mininet, qui est un outil libre et gratuit. Mininet est un émulateur réseau qui permet à l'utilisateur de créer un réseau composé de ponts, de liens et de terminaux tous virtuels. Ce type de réseaux virtuels permet de facilement prototyper et tester un réseau avant son déploiement, il permet également de déployer les applications réelles dans les terminaux virtuels afin d'en vérifier le comportement une fois mises en réseau.

La topologie du réseau peut être automatiquement générée par MoBACT en utilisant l'interface de programmation en Python fournie par Mininet. Les ponts virtuels de Mininet supportent OpenFlow [MAB+08], ce qui en fait un choix populaire dans le domaine du SDN (*Software-Defined Networking*). La capacité de déployer les applications qui utiliseront le réseau réel sur le réseau virtuel rend l'utilisation de Mininet intéressante pour effectuer des tests d'intégration tôt dans le processus de développement et permet de mesurer le volume réel des données qui seront échangées. Les ponts virtuels de Mininet ne supportent pas de fonctionnalités TSN mais nous anticipons un probable support de SDN [NDR16] par le matériel TSN dans le futur, ce qui ferait de Mininet un outil important dans l'ingénierie de réseaux TSN.

NeSTiNg

Nous avons également sélectionné NeSTiNg [FHC+19], un modèle de simulation qui rend possible la simulation de réseaux TSN dans OMNeT++ en ajoutant des fonctionnalités définies dans les standards TSN aux ponts et aux terminaux déjà existants dans INET et en permettant de les configurer.

NeSTiNg supporte plusieurs des standards TSN les plus importants pour l'ordonnancement du trafic le plus critique : TAS (IEEE 802.1Qbv), CBS (IEEE 802.1Qav) et préemption de trames (IEEE 802.1Qbu). C'est un outil libre et gratuit qui permet d'exécuter des simulations de réseaux TSN et de recueillir automatiquement les statistiques de cette simulation, comme les latences de bout en bout, et des les afficher sous forme de graphe. Il est également possible pour l'utilisateur d'extraire ces statistiques et de produire ses propres analyses.

Récemment, NeSTiNg a été intégré, ainsi que des fonctionnalités TSN qu'il ne supportait pas, à INET. Ce n'était pas le cas pour la majorité de la durée de cette thèse, ce qui explique

	Fonctionnalités TSN	Capacités d'analyses	Tests applicatif	Facilité d'accès
Mininet	ø	ø	++	++
NeSTiNg	+	+	ø	++
RTaW-Pegase	++	++	ø	- -

TABLE 8.1 – Différences entre les outils sélectionnés pour la génération automatique de configuration.

l'utilisation de NeSTiNg dans ce manuscrit. Il est tout de même à noter que l'implémentation d'un générateur de modèle de simulation pour INET a été réalisée au sein du laboratoire de Thales Research & Technology (dans lequel cette thèse s'est partiellement déroulée) et a été ajoutée à MoBACT.

RTaW-Pegase

Pour finir, nous avons sélectionné RTaW-Pegase comme troisième outil de conception. RTaW-Pegase est un autre simulateur de réseaux TSN qui supporte davantage de fonctionnalités TSN que NeSTiNg et qui propose plus de possibilités d'analyse des résultats de simulation.

Après avoir exécuter la simulation d'un réseau avec RTaW-Pegase, l'utilisateur peut visualiser les pires latences de bout en bout qui ont été observées lors de la simulation pour chaque flux de données. Il est également possible d'afficher des diagrammes de Gantt représentant le parcours des trames dans le réseau et leur moment d'arrivée dans chaque nœud du réseau. Ces diagrammes de Gantt permettent de visualiser le pire cas rencontrés lors de la simulation.

Contrairement à NeSTiNg, il n'est pas possible de visualiser graphiquement le déroulement de la simulation, ce qui peut permettre de facilement détecter une erreur dans le modèle (un flux qui n'aurait pas le bon chemin par exemple) ou un blocage dans un nœud des ponts du réseau. En revanche, RTaW-Pegase offre également la possibilité de calculer (par une approche basée sur le *network calculus*), et non de simuler, la latence de bout en bout de chaque flux de données dans le pire cas possible. Cette analyse est séparée de la simulation et permet à l'utilisateur de connaître avec certitude la borne que ne pourra dépasser chaque flux de données. Cette fonctionnalités est très intéressante car la simulation n'offre aucune garantie que ce pire cas théorique sera rencontré lors de son exécution alors qu'il est critique de garantir que les échéances des flux de données critiques sont toujours respectées.

RTaW-Pegase est un outil commercial et requiert une licence pour son utilisation, ce qui le rend beaucoup moins accessible que les outils académiques présentés précédemment.

Comparaison des outils sélectionnés

La table 8.1 regroupe les trois outils de conception que nous avons sélectionnés pour la génération automatique de configuration afin d'illustrer notre approche et résume leurs différences selon les quatre critères discutés : les fonctionnalités TSN supportées, les capacités d'analyse des résultats produits, la possibilité de déployer les applications réelles pour valider leur comportement et leur facilités d'accès.

8.1.2 Matériel

La génération de modèle de simulation et d'analyse permet de valider la configuration produite avant de la déployer sur le réseau réel mais cette dernière étape n'est pas a négliger. La création de fichier de configuration pour du matériel réseau n'est pas une tâche simple et c'est un processus fortement sujet à l'erreur humaine.

C'est pourquoi notre approche vise également à produire ces fichiers de configuration automatiquement, à partir du même modèle et de la configuration des mécanismes d'ordonnancement. Cette génération permet de conserver l'assurance de la cohérence entre la configuration déployée et les modèles de simulation.

Plusieurs formats existent pour ces fichiers de configuration : les standards TSN définissent un format utilisant les modèles de données YANG [M.10] et la plupart des constructeurs de matériel implémentent leur propre outil de déploiement de configuration. Dans notre cas, nous avons utilisé du matériel de la marque NXP [21] qui implémente un outil spécifique à son matériel.

8.1.3 Documentation

Enfin, un aspect à ne pas négliger est la documentation du réseau. La documentation permet d'avoir un moyen de donner accès à une version facilement compréhensible par les concepteurs du réseau.

Il est important de pouvoir vérifier que la topologie du réseau a été correctement modélisée et qu'elle correspond bien à la volonté de l'architecte réseau, ce qui peut être fait grâce à la génération d'une image d'un graphe de cette topologie. La génération d'une documentation complète du réseau est également intéressante afin de pourvoir partager l'architecture du réseau, sa topologie, la définition de ses flux de données et de sa configuration dans une forme intelligible.

8.2 Les capacités de MoBACT

MoBACT (*Model-Based Automatic Configuration for TSN*) est l'outil de génération de modèle de simulation et d'analyse, de fichiers de configuration pour du matériel et de documentation que nous avons implémenté afin de mettre notre approche en pratique.

Cet outil est implémenté en Java et permet dans un premier temps de compléter le modèle de la topologie réseau et des flux de données avec la configuration du TAS avant de passer à l'étape de génération.

8.2.1 TSNSched

Le rôle de l'étape de complétion de modèle est d'insérer des données dans un modèle de réseau TSN créé lors de l'étape précédente qui sont fastidieuses à saisir ou très compliquées à calculer pour l'utilisateur.

Cette étape peut, par exemple, générer automatiquement les adresses MAC des terminaux, puisque avoir une adresse réelle n'a pas d'importance pour un réseau simulé, utiliser des valeurs par défaut pour les bandes passantes ou créer automatiquement les éléments représentant les ports des nœuds en se basant sur les modèles des liens les reliant. Cette partie de la complétion n'a pour seul rôle que de simplifier et d'accélérer la création des modèles.

21. https://www.nxp.com/design/software/qoriq-developer-resources/layerscape-ls1028a-reference-design-board:LS1028ARDB

L'intérêt principal de l'étape de complétion de modèles est la synthèse de configuration du TAS, c'est à dire des GCL de chaque port de sortie du réseau, comme présenté dans le chapitre 2. La synthèse de cette configuration est en grande partie faite par un outil externe intégré à MoBACT : TSNsched [SSN19], également implémenté en Java ce qui facilite grandement son intégration dans MoBACT. De plus TSNsched est, d'après ses auteurs, le premier outil de génération automatique de configuration pour le TAS qui soit libre et en source ouverte. Cet outil peut synthétiser la configuration des fenêtres temporelles qui seront utilisées par les flux de données critiques en fonction des données contenues dans le modèle du réseau donné en entrée à MoBACT. TSNsched utilise un système de contraintes et un solveur SMT, z3 [dMB08], pour synthétiser, si une solution existe, la configuration de ce mécanisme d'ordonnancement de trafic de façon à pouvoir garantir le respect des exigences des flux de données critiques en matières de latence de bout en bout et de gigue maximale.

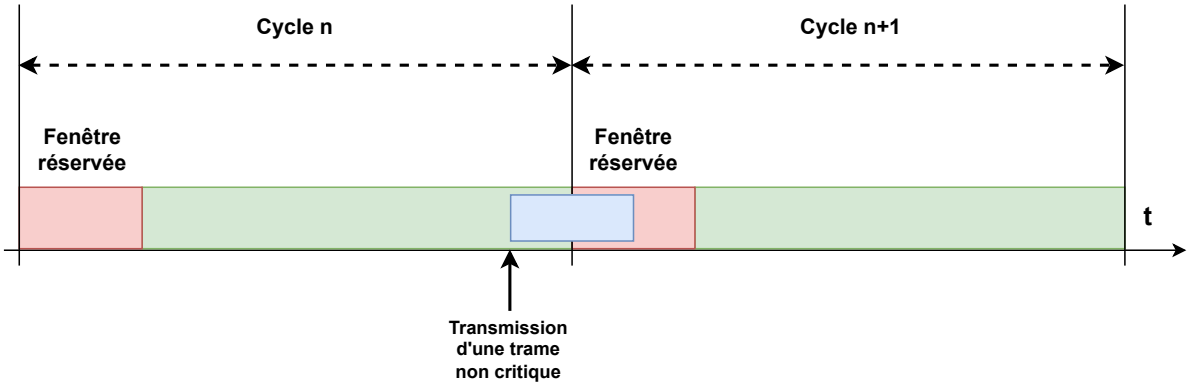


FIGURE 8.1 – Schéma présentant un exemple de blocage d'une fenêtre réservée par la transmission d'une autre trame en l'absence de *guard band*.

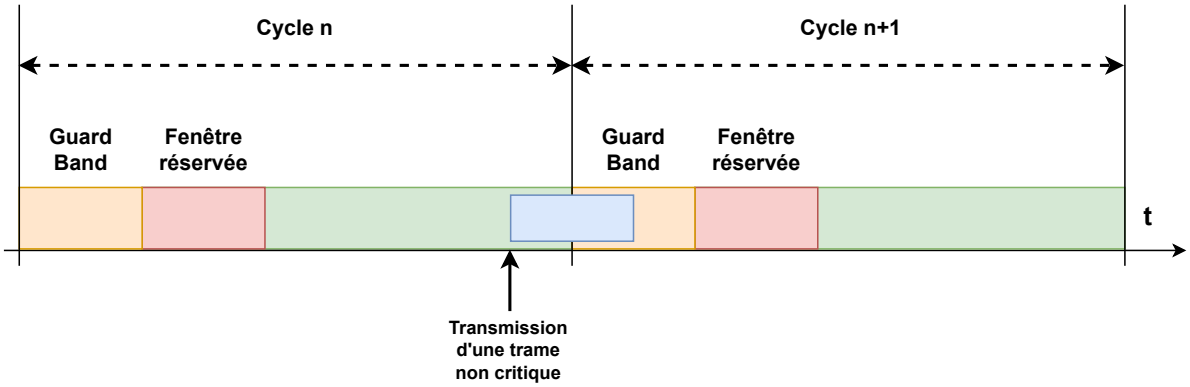


FIGURE 8.2 – Schéma présentant la protection apportée par l'utilisation d'une *guard band* pour protéger une fenêtre réservée à la transmission du trafic critique.

TSNsched calcule donc le moment du début d'une fenêtre temporelle réservée au trafic critique et sa durée tout en s'assurant qu'une durée suffisante reste disponible avant cette fenêtre pour pouvoir y insérer une *guard band*. Une *guard band* est une fenêtre temporelle placée juste avant une fenêtre réservée pour la transmission du trafic critique et pendant laquelle aucune transmission n'est autorisée à commencer, sauf dans le cas évoqué dans la section 2.2.3. Le rôle d'une *guard band* est de garantir qu'aucune transmission ne pourra empiéter sur la fenêtre tem-

porelle réservée au trafic critique. Cela assure donc que même la transmission d'une trame de taille maximale ne pourra jamais bloquer la transmission des trames appartenant aux flux de données critiques. La figure 8.1 illustre le phénomène de blocage d'une fenêtre réservée au trafic critique par la transmission d'une trame non critique en l'absence d'utilisation d'une *guard band*. La figure 8.2 présente, dans les mêmes conditions que pour la figure précédente, la protection apportée par l'utilisation d'une *guard band*.

Après que TSNsched ait synthétisé la durée du cycle, le positionnement et la durée des fenêtres réservées au trafic critique, notre travail consiste à y ajouter les fenêtres de *guard band* puis à créer des fenêtres pendant lesquelles le reste du trafic pourra être transmis pendant le temps restant du cycle. Le système de contraintes de TSNsched permet de garantir que pour chaque fenêtre protégée, il y suffisamment de temps avant celle-ci pour y insérer une fenêtre de *guard band*.

TSNsched essaye d'optimiser le positionnement des fenêtres réservées au trafic critique en faisant en sorte de les aligner de façon à ce que le moment où une trame critique est réceptionnée par un pont coïncide avec le moment où débute la fenêtre de transmission qui lui est réservée. Cela permet de minimiser le temps d'attente des trames critiques dans chaque pont et donc de minimiser la latence de bout en bout. Une fois toutes les fenêtres crées, celles pour le trafic critique, pour les *guard band* et pour le reste du trafic, la configuration du TAS est complète et peut être insérée dans le modèle pour servir ensuite à la génération des fichiers utilisés par les outils de simulation et d'analyse.

Cette stratégie de configuration du TAS permet de minimiser au maximum la latence des flux qui bénéficient d'une fenêtre protégée mais a souvent pour inconvénient la multiplication des fenêtres protégées et donc la multiplication des fenêtres de *guard band*, ce qui a tendance à gâcher une partie des ressources du réseau en bande passante. Si plusieurs flux de données sont disponibles pour l'émission sur un même port au même moment, TSNsched regroupe leurs fenêtres protégées en une seule d'une durée suffisante pour garantir la transmission des toutes ces trames.

8.2.2 Productions de MoBACT

MoBACT, notre outil de génération de configuration, prend les modèles créés lors de l'étape de modélisation, dans le format de notre syntaxe XML, comme entrée, comme le rappelle la figure 8.3. L'utilisateur peut ensuite spécifier quelles sont les cibles vers lesquelles générer les fichiers de configuration et s'il faut passer par l'étape de complétion de modèles.

Après avoir extrait les données du modèle, MoBACT génère un ensemble de fichiers contenant de la documentation sur le réseau et sa configuration, les fichiers générés pour les cibles sélectionnées parmi les trois actuellement supportées, Mininet, NeSTiNg et RTaW-Pegase, ainsi que les fichiers de configuration dans un format spécifique au modèle de pont TSN que nous utilisons au sein de notre laboratoire.

La documentation est générée sous la forme d'un ensemble de fichiers HTML. Un fichier est généré pour chaque élément du réseau et contient toutes les données concernant cet élément ainsi que ses liens avec les autres éléments du réseau. Ce type de documentation est utile pour facilement consulter et partager des informations dans un format plus simple qu'un fichier de modèle ; la figure 8.4 présente un exemple de la documentation générée pour le pont *pont1*.

Pour Mininet, la topologie du réseau à émuler peut être spécifiée à l'aide d'une interface de programmation en Python, comme présenté dans le listing 8.1. À l'aide des données contenues dans le modèle, MoBACT peut générer un fichier contenant sa topologie en utilisant cette interface de programmation. Le fichier contient l'instanciation de chaque élément du réseau

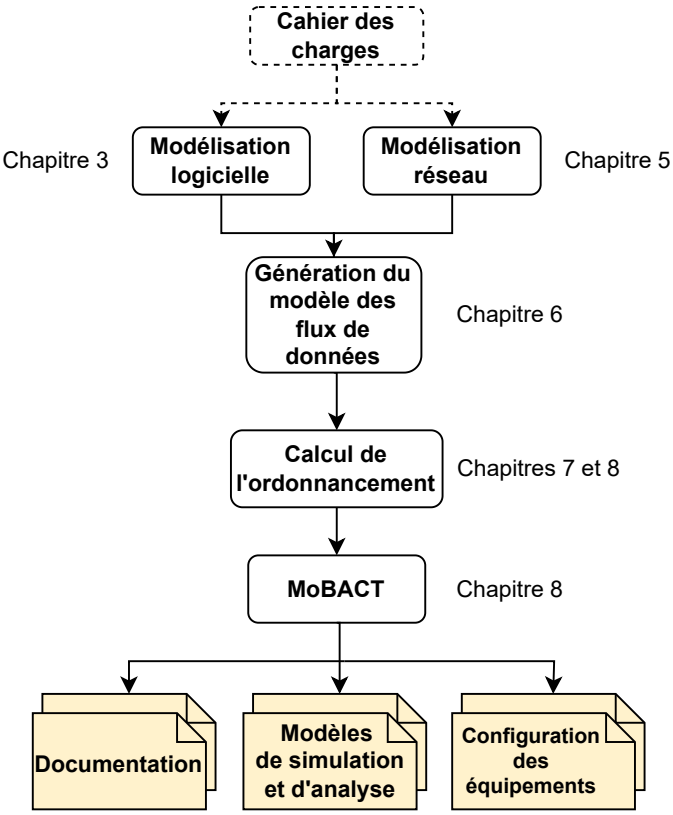


FIGURE 8.3 – Schéma représentant les différentes étapes de notre approche.

virtuel émulé par Mininet : les terminaux, les ponts et les liens.

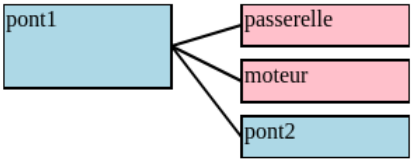
Pour NeSTiNg, un ensemble de fichiers est nécessaire pour décrire le topologie du réseau, sa configuration et pour définir les paramètres de la simulation. La topologie du réseau est définie par un fichier NED (*Network Description*), qui est un format spécifique à OMNeT++ permettant de déclarer les différents nœuds d'un réseau et les liens qui les relient. Un extrait d'un tel fichier est présenté par le listing 8.2. La configuration du réseau est répartie en plusieurs fichiers XML permettant d'établir les tables de commutation des ponts, la configuration des mécanismes d'ordonnancement de trafic et l'instanciation des flux de données. Les paramètres de la simulation sont définis dans un fichier spécifique au format INI.

Le listing 8.3 présente l'instanciation des flux de données du système dans la syntaxe XML utilisée par OMNeT++, le listing 8.4 présente l'initialisation des flux de données émis et reçu par le temrinal *controle* dans un fichier au format INI. Ces deux listings ont été générés à partir des éléments du modèle du réseau, définition et instanciation, présentés dans les listings 5.7 et 5.8.

Le listing 8.5 présente un extrait de la configuration du TAS utilisée pour le flux de données *consigne* au niveau du pont *pont1*.

index

Switch: pont1



pont1 has the following ports:

- eth0:
  - Connected to: passerelle (propagation delay = 0.001us)
  - Bandwidth = 100 Mbps
- eth1:
  - Connected to: moteur (propagation delay = 0.001us)
  - Bandwidth = 100 Mbps
  - TAS configuration:
    - Cycle duration = 30000us
    - Time slot 1: duration = 11008us, priorities: [6, 5, 4, 3, 2, 1, 0]
    - Time slot 2: duration = 10us, priorities: []
    - Time slot 3: duration = 10us, priorities: [7]
    - Time slot 4: duration = 18972us, priorities: [6, 5, 4, 3, 2, 1, 0]
- eth2:
  - Connected to: pont2 (propagation delay = 0.001us)
  - Bandwidth = 100 Mbps
  - TAS configuration:
    - Cycle duration = 30000us
    - Time slot 1: duration = 15994us, priorities: [6, 5, 4, 3, 2, 1, 0]
    - Time slot 2: duration = 10us, priorities: []
    - Time slot 3: duration = 10us, priorities: [7]
    - Time slot 4: duration = 13986us, priorities: [6, 5, 4, 3, 2, 1, 0]

pont1 is on the path of the following streams:

- conn_passerelle_passerelle1_ordre_emtr_pe_to_mission1
- consigne_conn_controle1_consigne_moteur_emtr_pe_to_moteur1
- etat_conn_moteur1_etat_emtr_pe_to_controle1

FIGURE 8.4 – Exemple de documentation produite par MoBACT pour le *pont1*.

```
[...]
class MyTopo (Topo):
  def build (self):

    #Add Host
    controle= self.addHost ('controle' ,ip='192.168.0.1/24')
    moteur= self.addHost ('moteur' ,ip='192.168.0.2/24')
    mission= self.addHost ('mission' ,ip='192.168.0.3/24')
    passerel1= self.addHost ('passerel1' ,ip='192.168.0.4/24')

    #Add Switch
    pont1= self.addSwitch ('pont1', dpid = '00:00:00:00:00:00:01')
    pont2= self.addSwitch ('pont2', dpid = '00:00:00:00:00:00:02')

    #Add Link
    self.addLink (passerel1,pont1, intf=TCIntf, params1={ 'ip' :'
       ↪ 192.168.0.4/24'})
    self.addLink (moteur,pont1, intf=TCIntf, params1={ 'ip' :'
       ↪ 192.168.0.2/24'})
    self.addLink (controle,pont2, intf=TCIntf, params1={ 'ip' :'
       ↪ 192.168.0.1/24'})
    self.addLink (mission,pont2, intf=TCIntf, params1={ 'ip' :'
       ↪ 192.168.0.3/24'})
    self.addLink (pont1,pont2)

topos = { 'mytopo': (lambda : MyTopo())}
```

Listing 8.1 – Example de spécification de topologie pour Mininet.

```
[...]
network app1_sr{
  types:
    channel controle_to_pont2 extends DatarateChannel {
      delay = 0.001us;
      datarate = 100Mbps;
    }

    [...]

  submodules:
    pont2: VlanEtherSwitchPreemptable {
      gates:
        ethg[3];
    }

    [...]

  connections:
    controle.ethg[0] <--> controle_to_pont2 <--> pont2.ethg[0];
    [...]
```

Listing 8.2 – Extrait d'un fichier NED déclarant la topologie du réseau pour NeSTiNg.

```
<schedules>
  <!-- conn_passerelle_passerelle1_ordre_emtr_pe_to_mission1-->
  <datagramSchedule id="0" baseTime="2.0us" cycleTime="180000.0us">
    <event payloadSize="60B" destAddress="app1_sr.mission" destPort="
       ↪ 1000" pcp="5" vid="1" timeInterval="0ms"/>
  </datagramSchedule>

  <!-- consigne_conn_controle1_consigne_moteur_emtr_pe_to_moteur1 -->
  <datagramSchedule id="1" baseTime="11002.0us" cycleTime="30000.0us">
    <event payloadSize="38B" destAddress="app1_sr.moteur" destPort="1000
       ↪ " pcp="7" vid="1" timeInterval="0ms"/>
  </datagramSchedule>

  <!-- etat_conn_moteur1_etat_emtr_pe_to_controle1 -->
  <datagramSchedule id="2" baseTime="26004.0us" cycleTime="30000.0us">
    <event payloadSize="44B" destAddress="app1_sr.controle" destPort="
       ↪ 1000" pcp="7" vid="1" timeInterval="0ms"/>
  </datagramSchedule>

  <!-- ordre_mission_mission1_ordre_emtr_pe_to_controle1 -->
  <datagramSchedule id="3" baseTime="10002.0us" cycleTime="30000.0us">
    <event payloadSize="60B" destAddress="app1_sr.controle" destPort="
       ↪ 1001" pcp="6" vid="1" timeInterval="0ms"/>
  </datagramSchedule>
</schedules>
```

Listing 8.3 – Instanciation des flux de données dans NeSTiNg.

```
app1_sr.controle.numApps = 3
app1_sr.controle.app[0].typename = "UdpSink"
app1_sr.controle.app[0].localPort = 1000
app1_sr.controle.app[1].typename = "UdpSink"
app1_sr.controle.app[1].localPort = 1001
app1_sr.controle.app[2].typename = "UdpScheduledTrafficApp"
app1_sr.controle.app[2].trafficGenerator.localPort = 1002
app1_sr.controle.app[2].scheduleManager.initialAdminSchedule = xmldoc("
   ↪ xml/app1_sr_flows.xml", "/schedules/datagramSchedule[@id='1']")
```

Listing 8.4 – Initialisation des flux de données du terminal *controle* dans NeSTiNg.

```
<schedules>
  <switch name="pont1">
    [...]
    <port id="1">
      <schedule cycleTime="30000us">
        <entry>
          <length>11008us</length>
          <bitvector>01111111</bitvector>
        </entry>
        <entry>
          <length>10us</length>
          <bitvector>00000000</bitvector>
        </entry>
        <entry>
          <length>10us</length>
          <bitvector>10000000</bitvector>
        </entry>
        <entry>
          <length>18972us</length>
          <bitvector>01111111</bitvector>
        </entry>
      </schedule>
    </port>
    [...]
</schedules>
```

Listing 8.5 – Extrait de la configuration du TAS utilisée pour le flux de données *consigne* dans NeSTiNg.

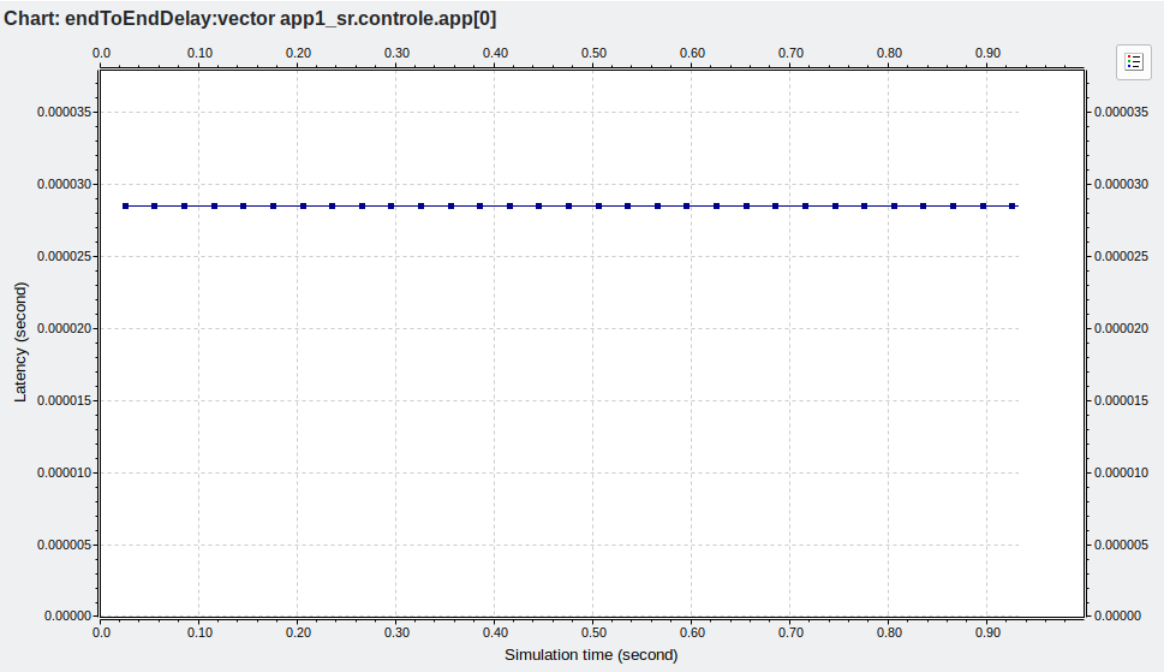


FIGURE 8.5 – Courbe de la latence de bout en bout du flux de données *consigne* en utilisant le TAS avec NeSTiNg.

Au cours du déroulement de la simulation, un ensemble de données sont automatiquement collectées par OMNeT++ et permettent d'afficher des graphes comme celui de la latence de bout en bout de chaque flux de données. La figure 8.5 présente, le graphe de latence de bout en bout du flux de données *consigne* obtenu avec utilisation du TAS. Les résultats correspondent au comportement attendu : l'utilisation du TAS permet d'avoir une latence de bout en bout faible, sa valeur est de 28,48 µs, et constante, respectant ainsi les exigences temps réel.

Le listing 8.6 présente un extrait du modèle de simulation utilisé par RTaW-Pegase contenant la définition de la topologie du réseau au format CSV.

Le listing 8.7 présente la configuration du TAS qu'il faut déployer sur le port *eth1* du pont *pont1* dans le format supporté par le matériel de la marque NXP, un fichier tel que celui-ci est généré pour chaque port de chaque pont du réseau. Le listing 8.8 contient la suite de commandes à utiliser pour déployer la configuration du CBS sur le pont *pont1*, un fichier contenant une telle suite de commandes est généré pour chaque pont du réseau.

```
[Nodes]
[Name]
controle
moteur
mission
passerelle
[EthernetTopology]
[Name];Topology
[NodeSet];app1_sr_NodeSet
[Routers]
[Name];[Switching (us)];[Memory (byte)]
pont1;2.0;
pont2;2.0;
[Wired Links]
[Name];[End1];[Interface1];[End2];[Interface2];[Speed (Mbit/s)];[Bit
    ↪ Traversal Time]
passerelle_to_pont1;passerelle;P1;pont1;P1;100.0;0.001
moteur_to_pont1;moteur;P1;pont1;P2;100.0;0.001
controle_to_pont2;controle;P1;pont2;P1;100.0;0.001
mission_to_pont2;mission;P1;pont2;P2;100.0;0.001
pont1_to_pont2;pont1;P3;pont2;P3;100.0;0.001
```

Listing 8.6 – Extrait du modèle de simulation utilisé par RTaW-Pegase au format CSV.

```
t0  01111111b 11008000
t1  00000000b 10000
t2  10000000b 10000
t3  01111111b 18972000
```

Listing 8.7 – Configuration du TAS à déployer sur le port *eth1* du pont *pont1* dans le format spécifique aux matériel de la marque NXP.

```
tc qdisc add dev eth0 root handle 1: mqprio num_tc 8 map 0 1 2 3 4 5 6 7
    ↪  hw 1
tc qdisc add dev eth1 root handle 1: mqprio num_tc 8 map 0 1 2 3 4 5 6 7
    ↪  hw 1
tc qdisc add dev eth2 root handle 1: mqprio num_tc 8 map 6 hw 0.11
tc qdisc add dev eth2 root handle 1: mqprio num_tc 8 map 5 hw 0.42
```

Listing 8.8 – Suite de commances permettant le déploiement de la configuration du CBS à déployer sur le pont *pont1*.

Chapitre 9

Expérimentations

L'objectif de ce chapitre est de démontrer l'intérêt de notre approche de modélisation et de génération, en termes de gain de productivité et de confiance dans la configuration générée, dans différents cas.

Nous présenterons donc la mise en pratique de notre approche dans plusieurs cas d'évolution du système : lorsque la nature des communications est modifiée, par exemple en passant à des communications signées, et dans le cas où un nouveau composant est ajouté au système, générant ainsi de nouvelles transmissions.

Nous appliquerons ensuite notre méthode de calcul de configuration du CBS à deux flux de données de type multimédia.

Nous étudierons finalement le cas d'un système totalement différent utilisant une topologie ayant plus de nœuds et sur laquelle sont transmis plus de flux de données.

9.1 Utilisation de communications signées

Cette section reprend le système présenté dans le chapitre 4, dont nous rappelons la topologie dans la figure 9.1, et démontre l'intérêt de notre approche dans le cas d'une évolution du système. Une évolution mineure d'un système peut avoir des conséquences importantes sur ce qui est produit à partir des modèles, que ce soit le code technique des applications ou la configuration du réseau.

Le listing 9.1 présente le plan d'application, dont le concept est défini dans le chapitre 3, utilisé par l'application servant d'exemple de base au cours de ce manuscrit. Ce listing comporte la déclaration des différents composants qui forment l'application et la déclaration des connecteurs *consigne_conn* et *etat_conn* qui permettent l'échange des flux de données de contrôle-commande.

Le listing 9.2 présente le plan d'allocation, également défini dans le chapitre 3, utilisé par l'application servant d'exemple de base au cours de ce manuscrit. Ce listing comporte la configuration des fichiers exécutables qui contiennent les composants *Controle* et *Moteur* ainsi que la configuration de la politique technique d'exécution périodique qui s'applique au composant *Controle*.

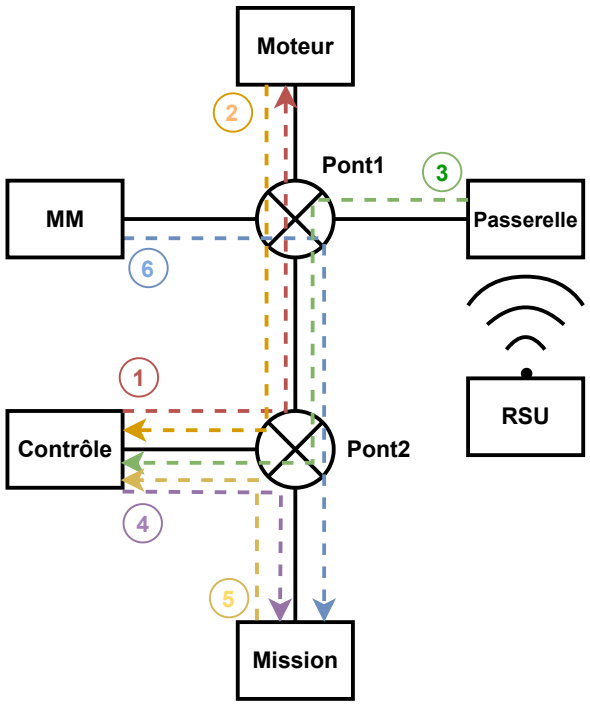


FIGURE 9.1 – Schéma de la topologie du système étudié.

```
<appliPlan name="app1">
  <appAssembly name="app1">
   <part name="controle1" ref="::model::detailed_comps::controle1"/>
   <part name="moteur1" ref="::model::detailed_comps::moteur1"/>
   <part name="passerelle1" ref="::model::detailed_comps::passerelle1"/>
   <part name="mission1" ref="::model::detailed_comps::mission1"/>
   <part name="boussole1" ref="::model::detailed_comps::boussole1"/>
   <part name="position1" ref="::model::detailed_comps::position1"/>
   <connection name="consigne_conn" ref="::pharos_lib::socket_comm::
   ↪ prs_socket_msg_cnt">
    <end name="controle1_consigne_moteur" part="controle1" port="
   ↪ consigne_moteur"/>
    <end name="moteur1_consigne" part="moteur1" port="consigne"/>
    <config def="default_priority" value="7"/>
   </connection>
   <connection name="etat_conn" ref="::pharos_lib::socket_comm::
   ↪ prs_socket_sd_cnt">
    <end name="moteur1_etat" part="moteur1" port="etat"/>
    <end name="controle1_etat_moteur" part="controle1" port="etat_moteur
   ↪ "/>
    <config def="default_priority" value="7"/>
   </connection>
    [...]
  </appAssembly>
</appliPlan>
```

Listing 9.1 – Plan d'application utilisé par le système servant d'exemple de base.

```
<allocPlan name="app1_alloc">
  <appliPlanRef ref="::model::app1"/>
  <environmentRef ref="::model::topologie1"/>
  <compNode def="::pharos_lib::rsc::prs_process" lang="::ucm_lang::cpp::
   ↪ CPP11_typed" name="controle1">
   <compInstRef ref="controle1"/>
   <execRsc def="::pharos_lib::rsc::prs_task" name="controle1_task"/>
   <config def="udp_port" value="1234"/>
   <rscConfig def="hw_allocation" value="::model::topologie1::controle"/
   ↪ >
  </compNode>
  <compNode def="::pharos_lib::rsc::prs_process" lang="::ucm_lang::cpp::
   ↪ CPP11_typed" name="moteur1">
   <compInstRef ref="moteur1"/>
   <config def="udp_port" value="1235"/>
   <rscConfig def="hw_allocation" value="::model::topologie1::moteur"/>
  </compNode>
  [...]
  <policyConf name="exec_periodique_controle1" policy="exec_periodique">
   <compInstRef ref="controle1"/>
   <rscConfig def="task_alloc" value="controle1.controle1_task"/>
  </policyConf>
  [...]
</allocPlan>
```

Listing 9.2 – Plan d'allocation du système de base.

```
<allocPlan name="app1_rsa_alloc">
  <refines ref="::model::app1_alloc"/>
  <appliPlanRef ref="::model::app1"/>
  <environmentRef ref="::model::topologie1"/>
  <connConf name="consigne_rsa_conn">
    <connInstRef ref="consigne_conn"/>
    <refinedIn ref="::pharos_lib::auth_comm::prs_rsa4096_msg_cnt"/>
    <endConf name="controle1_consigne_moteur">
      <endInstRef ref="controle1_consigne_moteur"/>
      <config def="eid" value="1"/>
    </endConf>
  </connConf>
  <connConf name="etat_rsa_conn">
    <connInstRef ref="etat_conn"/>
    <refinedIn ref="::pharos_lib::auth_comm::prs_rsa4096_sd_cnt"/>
    <endConf name="moteur_etat">
      <endInstRef ref="moteur1_etat"/>
      <config def="eid" value="1"/>
    </endConf>
  </connConf>
</allocPlan>
```

Listing 9.3 – Plan d'allocation comporant le raffinement des connecteurs utilisés pour la transmission des flux *consigne* et *état* pour signer ces communications par RSA.

L'évolution présentée ici est le passage à des communications signées pour les deux flux de données entre les composants *contrôle* et *moteur*. Pour des raisons de sécurité, la décision a été prise par l'architecte système d'utiliser des communications signées par RSA [RSA78] avec une clé de taille 4096 bits pour les deux flux critiques échangés entre *contrôle* et *moteur*. C'est une décision qui peut être prise par l'architecte du système afin de sécuriser les communications de ces deux flux de données, qui sont les deux flux les plus critiques du système.

Le listing 9.3 présente l'unique changement qu'il est nécessaire de faire au modèle de l'architecture des applications pour utiliser des connecteurs engendrant des communications signées par RSA pour ces deux flux de données. Ce nouveau plan d'allocation raffine celui du listing 9.2 en modifiant les connecteurs utilisés. Les lignes 7 et 15 indiquent que les connecteurs utilisés pour ces communications dans le système de base sont raffinés pour utiliser à la place des connecteurs incorporant une signature RSA.

Ce changement d'architecture, aussi minime soit-il en termes de modélisation, va avoir un impact important sur la configuration du réseau qui sera générée et plus particulièrement sur la configuration des mécanismes d'ordonnancement de trafic. En effet, l'ajout d'une signature RSA à ces communications va grandement augmenter la taille des données qui seront envoyées à chaque message. Ce volume de données supplémentaires va invalider la configuration des mécanismes d'ordonnancement produite pour le système de base : dans le cas du TAS, la durée des fenêtres temporelles ne sera plus suffisante pour permettre la transmission de ces nouveaux messages plus longs, dans le cas du CBS, la proportion de bande passante allouée à ces communications ne sera plus suffisante pour un volume de données plus important.

```
1  [stream2]
2   name = "consigne_conn_controle1_consigne_moteur_emtr_pe_to_moteur1"
3   payload.value = 558
4   payload.unit = "B"
5   type = "Periodic_Stream"
6   period.value = 30000
7   period.unit = "us"
8   talker = ["::model::topologie1::controle_to_pont2.controle_eth0"]
9   listener = ["::model::topologie1::moteur_to_pont1.moteur_eth0"]
10  pcp = ["7"]
11  offset.value = 11002
12  offset.unit = "us"
13
14 [stream3]
15  name = "etat_conn_moteur1_etat_emtr_pe_to_controle1_etat_moteur"
16  payload.value = 564
17  payload.unit = "B"
18  type = "Periodic_Stream"
19  period.value = 30000
20  period.unit = "us"
21  talker = ["::model::topologie1::moteur_to_pont1.moteur_eth0"]
22  listener = ["::model::topologie1::controle_to_pont2.controle_eth0"]
23  pcp = ["7"]
24  offset.value = 26004
25  offset.unit = "us"
```

Listing 9.4 – Paramètres des flux de données *consigne* et *etat* avec l'utilisation de connecteurs signant les communications par RSA.

Le listing 9.4 présente les paramètres des deux flux de données impactés par ce changement d'architecture obtenus par le processus automatique présenté au chapitre 6. Le seul paramètre dont la valeur est différente de celle obtenue pour le système de base est la taille de la charge utile qui est maintenant bien plus grande.

Pour ces deux flux de données, dont la taille de la charge utile dans l'itération du système n'utilisant pas de communications signées est de respectivement 38 octets et 44 octets, l'augmentation de la taille de la charge utile est de 520 octets. Comme expliqué dans l'annexe B, la taille de la signature est de 512 octets et 8 autres octets sont ajoutés à la charge utile afin d'identifier l'application destinataire, ce qui porte la taille de la charge utile à respectivement 558 octets et 564 octets.

L'augmentation de la taille de la charge utile nécessite que la configuration du TAS soit calculée à nouveau, la configuration obtenue pour le système de base ne pouvant plus être utilisée. Cette ancienne configuration n'est plus valide car la taille de fenêtres protégées n'est plus suffisante mais également parce que la taille de fenêtres de *guard band* ne le sont plus non plus, les deux ayant une durée trop faible pour permettre la transmission de ces nouvelles trames dont la taille a augmentée.

```
Switch: pont1 (::model::topologie1::pont1)
 [...]
 Port eth1: pont1_eth1
  TAS configuration:
   Cycle duration: 30000us
   Time slot 0: duration: 11056us, priorities: [6, 5, 4, 3, 2, 1, 0]
   Time slot 1: duration: 52us, priorities: []
   Time slot 2: duration: 52us, priorities: [7]
   Time slot 3: duration: 18840us, priorities: [6, 5, 4, 3, 2, 1, 0]
 [...]

Switch: pont2 (::model::topologie1::pont2)
 [...]
 Port: eth2: pont2_eth2
  TAS configuration:
   Cycle duration: 30000us
   Time slot 0: duration: 11002us, priorities: [6, 5, 4, 3, 2, 1, 0]
   Time slot 1: duration: 52us, priorities: []
   Time slot 2: duration: 52us, priorities: [7]
   Time slot 3: duration: 18894us, priorities: [6, 5, 4, 3, 2, 1, 0]
 [...]
```

Listing 9.5 – Configuration du TAS sur le chemin du flux de données *consigne* dans l'itération du système utilisant le chiffrage RSA.

Le listing 9.5 présente la configuration du TAS le long du chemin du flux de données *consigne* pour cette itération du système, utilisant des communications signées, dans le format que nous utilisons pour la documentation du réseau générée par MoBACT.

La figure 9.2 présente les résultats de simulation obtenus avec NeSTiNg pour le flux de données *etat*. Ce flux de données appartient à la classe de trafic contrôle-commande et il est donc géré par le TAS. Sa latence de bout en bout est constante grâce aux fenêtres protégées et sa valeur est de 153 µs.

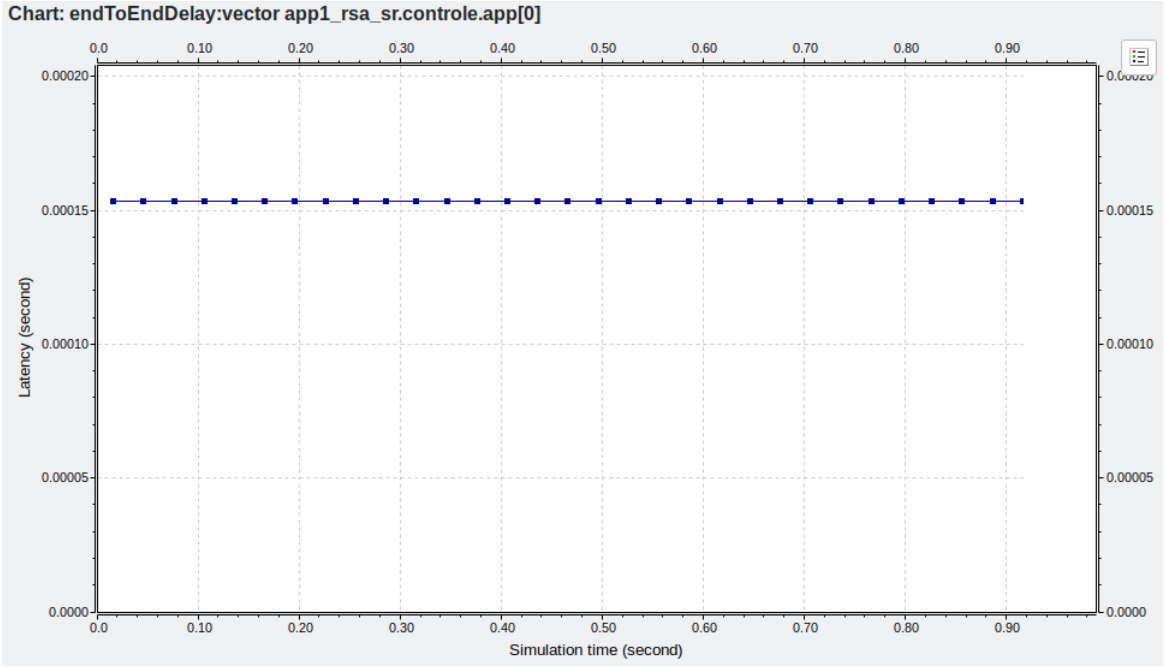


FIGURE 9.2 – Courbe de la latence de bout en bout du flux de données *etat* en utilisant le TAS avec NeSTiNg dans le cas des communications signées.

9.2 Ajout d'un composant au système

Une autre possibilité d'évolution du système est l'ajout de nouveaux composants permettant d'étendre les fonctionnalités du système. À des fins de test, l'architecte du système a décidé de mettre en place un moyen de surveiller les deux flux de données les plus critiques, les flux *consigne* et *etat* échangés entre les nœuds *controle* et *moteur*. Pour ce faire, un nouveau composant logiciel est ajouté au système : le composant *observateur*. Ce nouveau composant est déployé sur le nœud déjà existant *passerelle* puisque son objectif est de surveiller deux flux de données puis de les transmettre vers des éléments extérieurs au système par communication sans-fil.

```
1 <compType name="observateur">
2   <port bindings="consigne_b" name="observation_c1" type="::ucm_core::
      ↪ messages::msg_rcvr_pt"/>
3   <port bindings="etat_b" name="observation_m1" type="::ucm_ext_interac
      ↪ ::shared_data::sd_reader_pt"/>
4 </compType>
```

Listing 9.6 – Déclaration du type de composant *Observateur*.

Le listing 9.6 présente la déclaration du type de ce nouveau composant. Il possède deux ports qui recevront les messages correspondant aux deux flux de données observés *consigne* et *etat*.

La figure 9.3 décrit l'architecture logicielle permettant l'échange des flux de données *consigne* et *etat* entre les nœuds *controle* et *moteur*. Elle décrit également la moyen d'observation de ces deux flux de données : le nouveau composant *observateur* et ses deux ports relié aux connecteurs assurant les communications à observer.

Le listing 9.7 présente le plan d'application du système utilisant le composant d'observation.

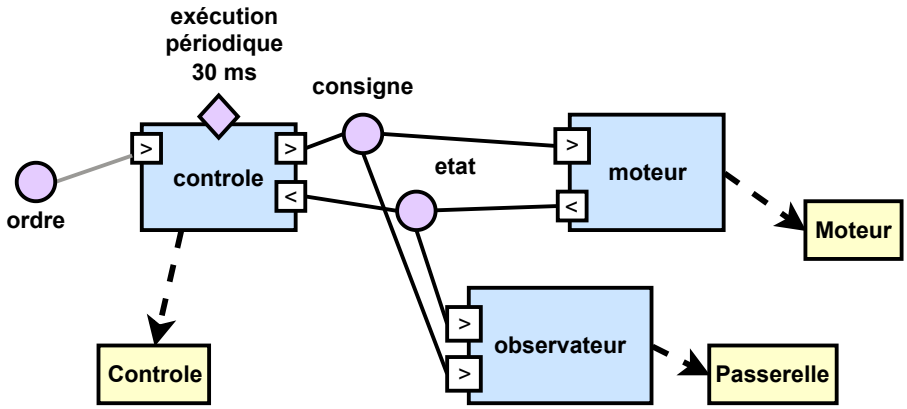


FIGURE 9.3 – Schéma des composants et des connecteurs dans la version du système utilisant un composant observateur.

```
<appliPlan name="app2">
  <appAssembly name="app2">
   <refines ref="::model::app1::app1"/>
   <part name="observateur1" ref="::model::detailed_comps::observateur1"
   ↪ />
   <connection name="etat_conn" ref="::pharos_lib::socket_comm::
   ↪ prs_socket_sd_cnt">
    <refines ref="::model::app1::app1.etat_conn"/>
    <end name="observateur1_etat" part="observateur1" port="
   ↪ observation_m1"/>
   </connection>
   <connection name="consigne_conn" ref="::pharos_lib::socket_comm::
   ↪ prs_socket_msg_cnt">
    <refines ref="::model::app1::app1.consigne_conn"/>
    <end name="observateur1_consigne" part="observateur1" port="
   ↪ observation_c1"/>
   </connection>
  </appAssembly>
</appliPlan>
```

Listing 9.7 – Plan d'application du système utilisant un composant d'observation.

Ce plan d'application raffine celui du système de base en y rajoutant la déclaration de l'utilisation du composant *observateur1* et en ajoutant ce composant aux extrémités du connecteur *consigne_conn*.

Pour résumer, la façon de modéliser l'ajout d'un composant d'observation de flux de données déjà existants en UCM est de déclarer un nouveau type de composant et de relier ses ports aux connecteurs déjà utilisés pour l'échange des flux de données observés.

Ces nouvelles connexions aux connecteurs vont engendrer de nouveaux flux de données puisque les connecteurs utilisés, venant de la bibliothèque Pharos présentée dans l'annexe B, sont des connecteurs *unicast* (ce choix a été discuté dans la section 5.5). De nouveaux flux de données entraînent un besoin de reconfigurer le réseau pour garantir que ces changements n'empêcheront pas le respect des exigences temps-réel du système.

```
[stream2]
 name = "consigne_conn_controle1_consigne_moteur_emtr_pe_to_observateur1
   ↪ "
 payload.value = 38
 payload.unit = "B"
 type = "Periodic_Stream"
 period.value = 30000
 period.unit = "us"
 talker = ["::model::topologie1::controle_to_pont2.controle_eth0"]
 listener = ["::model::topologie1::passerelle_to_pont1.passerelle_eth0"]
 pcp = ["0"]
 offset.value = 11002
 offset.unit = "us"

[stream4]
 name = "etat_conn_moteur1_etat_emtr_pe_to_observateur1"
 payload.value = 44
 payload.unit = "B"
 type = "Periodic_Stream"
 period.value = 30000
 period.unit = "us"
 talker = ["::model::topologie1::moteur_to_pont1.moteur_eth0"]
 listener = ["::model::topologie1::passerelle_to_pont1.passerelle_eth0"]
 pcp = ["0"]
 offset.value = 26004
 offset.unit = "us"
```

Listing 9.8 – Paramètres des flux de données d'observation des flux *consigne* et *etat*.

Le processus présenté dans le chapitre 6 nous permet de générer automatiquement le modèle des deux nouveaux flux de données. Les paramètres automatiquement calculés de ces flux sont présentés dans le listing 9.8. On peut remarquer que le niveau de priorité de ces deux nouveaux flux de données est différent de celui des flux qu'ils observent, et ce même si tous ces flux de données utilisent les mêmes connecteurs. Il est possible en UCM de spécifier des niveaux de priorités spécifiques à certaines communications.

En effet, ces deux nouveaux flux de données servant à l'observation des flux de données échangés entre *controle* et *moteur*, sans spécification particulière, auraient la même priorité que les flux observés. Il est donc possible en UCM de spécifier, au niveau de la configuration des connecteurs dans le plan d'allocation, des niveaux de priorité spécifiques à certaines configurations qui auront la précédence sur la valeur initiale.

Ces deux nouveaux flux de données n'étant que des flux d'observations, pouvant être utilisés à des fins de test, ils n'appartiennent pas à la classe de trafic contrôle-commande mais à la classe « meilleur effort » et ne bénéficient donc pas de fenêtres protégées dans la configuration du TAS.

9.3 Flux multimédia

Plusieurs types de trafic différents peuvent circuler en même temps sur un même réseau TSN. Comme présenté dans le chapitre 7, plusieurs mécanismes d'ordonnancement peuvent être utilisés conjointement afin de s'adapter à ces différents types de trafic de façon à ce que toutes

leurs exigences soient respectées. C'est le cas dans le système que nous avons présenté dans les chapitres précédents.

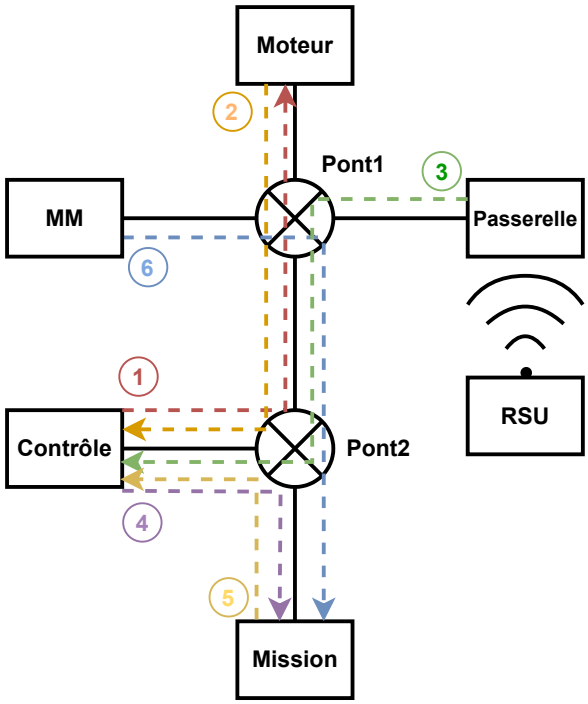


FIGURE 9.4 – Rappel de la topologie du système exemple.

Les flux de données de type multimédia du système sont produits par le terminal *MM* et transmis à destination du terminal *Mission*, comme indiqué par le chemin 6 de la figure 9.4.

Classe	Charge utile	C_i	T_i	D_i^{local}	D_i^{global}
A	325 B	26 µs	125 µs	1 ms	2 ms pour 2 sauts
B	325 B	26 µs	250 µs	10 ms	20 ms pour 2 sauts
BE	325 B	26 µs	125 µs	∅	∅

TABLE 9.1 – Classes de trafic différentes du contrôle-commande présentes dans le système.

La table 9.1 présente les différentes classes de trafic des flux de données autres que ceux de type contrôle-commande circulant sur le réseau. Ces classes de trafic forment un exemple typique de flux de données audio (classe A), vidéo (classe B) et « meilleur effort » (classe BE). Les paramètres C_i, T_i et D_i correspondent respectivement au temps de transmission d'une trame, à la période de transmission et à l'échéance que doivent respecter les trames de la classe i.

L'échéance à respecter est exprimée de façon locale et globale. En effet, notre approche de calcul de la configuration du CBS est une approche locale et elle nécessite donc une valeur locale de l'échéance. Un moyen simple d'obtenir cette valeur est de diviser la valeur de l'échéance globale par le nombre de sauts que doivent effectuer les flux de données. Le chemin de ces flux de données passant par deux ponts, la valeur de l'échéance locale est égale à l'échéance globale divisée par deux.

Dans le cas de ce système, nous considérons la présence de deux flux de données de classe A, un flux de données de classe B et un flux de données de classe BE. Comme présenté précédemment

dans ce chapitre, les paramètres concernant les flux de données appartenant à la casse de trafic contrôle-commande ont les valeurs suivantes : la durée de la fenêtre protégée est $L_{PW} = L_{TC} + L_{GB} = 10\ \mu s + 26\ \mu s = 36\ \mu s$ et la durée du cycle TAS est $L_{TAS} = 30000\ \mu s$.

Pour obtenir la configuration du CBS à utiliser pour la classe de trafic A, nous appliquons l'équation 7.7 pour obtenir $\alpha_A^+ \geq max(\frac{0{,}416}{0{,}9988}, \frac{26}{1000-26-26-36}) \times BW = max(0,4165, 0,0285) \times BW$. Cette équation nous donne la proportion minimum de bande passante qu'il est nécessaire de réserver afin de respecter l'échéance de la classe de trafic A. Nous choisissons donc la valeur $\alpha_A^+ = 0,42 \times BW$.

Une fois cette valeur calculée, il est nécessaire de s'assurer qu'elle respecte également la condition spécifiée par l'équation 7.8 : $0,42 \times BW \leq (1 - \frac{36}{30000}) \times BW = 0,42 \leq 0,9988$. La condition est respectée, ce qui signifie qu'il y a suffisamment de bande passante disponible pour que la transmission des flux de données de cette classe de trafic respecte son échéance, la valeur obtenue pour α_A^+ peut donc être utilisée.

Nous pouvons maintenant utiliser l'équation 7.9 afin d'obtenir la configuration du CBS à utiliser pour la classe de trafic B : $\alpha_B^+ \geq max(\frac{0{,}104}{0{,}9988}, \frac{0}{10000-26-26\times(1+\frac{0{,}42}{0{,}58})-26-36}) \times BW = max(0,104, 0) \times BW$. Nous choisissons donc la valeur $\alpha_B^+ = 0,11 \times BW$. Le numérateur du deuxième argument de la fonction $max()$ est 0 car il n'y a qu'un seul flux de données dans cette classe de trafic.

De la même façon que pour la valeur α_A^+, cette valeur doit respecter la condition exprimée par l'équation 7.10 : $0,11 \times BW \leq (1 - \frac{36}{30000}) \times BW - 0,42 \times BW = 0,9988 \times BW - 0,42 \times BW = 0,5788 \times BW$. La condition est respectée et la valeur α_B^+ peut être utilisée.

Les calculs que nous venons de présenter permettent d'obtenir la configuration du CBS à utiliser sur le port du *pont1* le reliant au *pont2*. Le même calcul doit être répété pour obtenir la configuration à utiliser sur le port du *pont2* le reliant au terminal *mission*.

Nous pouvons conclure de l'application de notre approche de calcul de la configuration du CBS que cet ensemble de flux de données est ordonnançable. Nous pouvons également noter que les classes de trafic A et B sont dominées par le taux d'utilisation de la bande passante plutôt que par leur échéances respectives.

9.4 Topologie plus complexe

Afin de mettre en pratique notre approche sur un système plus complexe que l'exemple utilisé jusqu'à maintenant, nous proposons un nouveau système utilisant une topologie plus complexe, de type partiellement maillée. Ce système fictionnel est censé être embarqué sur un drone possédant quatre moteurs ainsi que plusieurs systèmes permettant de déterminer son cap et sa position ce qui génèrera de plus nombreux flux de données. Le modèle complet de ce réseau (topologie, flux de données et exigences système) est disponible dans l'annexe A.3.

La figure 9.5 présente la topologie de ce système, qui est composée de six ponts et de douze terminaux. Le fonctionnement général du système est semblable à celui de l'exemple présenté dans les sections précédentes mais à une plus grande échelle. Le composant *passerelle* reçoit des consignes de destination par communications sans-fil et les transmet au composant *mission*. Ce composant effectue des calculs et transmet cette consigne aux différents composants *controle* afin qu'elle soit appliquée. Le drone possédant quatre moteurs, il y a quatre composants *controle*, chacun en charge d'un moteur. Afin d'effectuer ses calculs, le composant *mission* utilise les données transmises par les composants *boussole* et *position*. Tout comme dans le système exemple utilisé précédemment, les composants *moteur* répondent aux composants *controle* en transmettant leur état actuel.

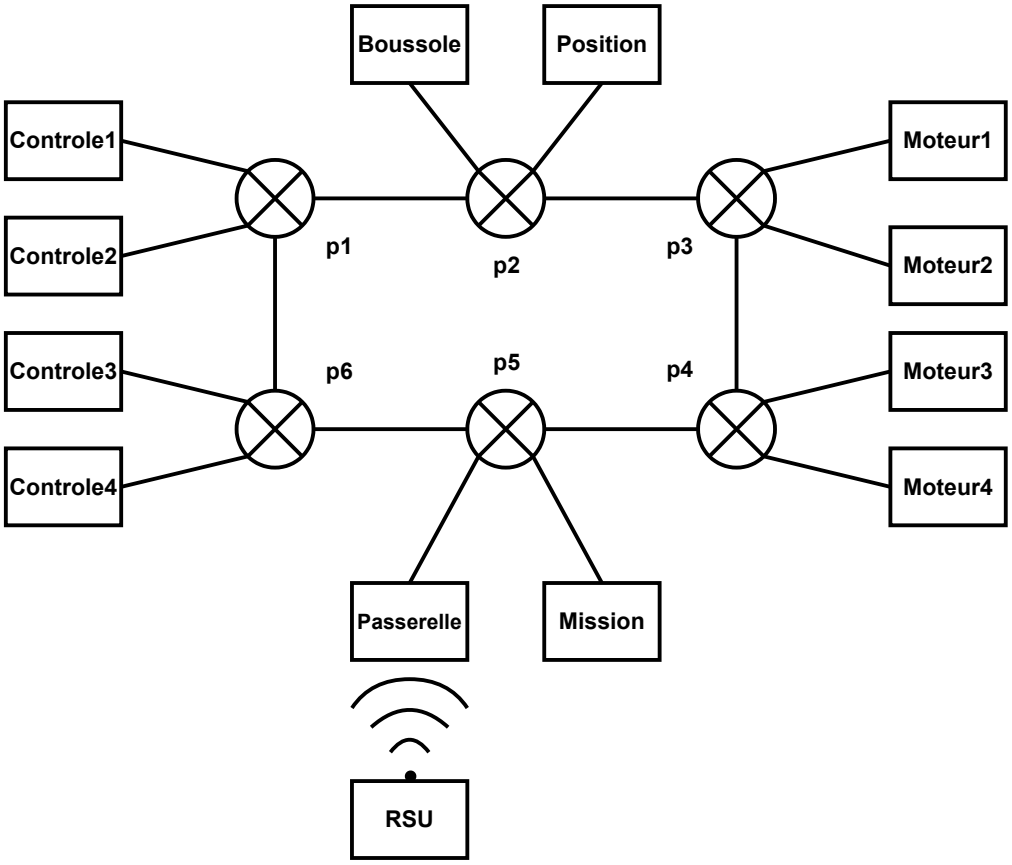


FIGURE 9.5 – Topologie plus complexe du système embarqué sur un drone.

Certains de ces composants ayant le même rôle et le même comportement que ceux utilisés dans le système exemple des sections précédentes, il est possible de bénéficier de l'approche de modélisation logicielle par composants d'UCM pour réutiliser certains composants et fortement limiter le temps de développement de ce nouveau modèle. Seuls les nouveaux composants comme *boussole* et *position* sont à concevoir avant de réaliser le nouveau plan d'allocation et la nouvelle topologie du réseau.

Une fois ce nouveau modèle créé, nous pouvons générer automatiquement le modèle des flux de données puis ajouter les exigences système qui s'appliquent à ces flux. Cette dernière étape représente la fin de la modélisation du système et il est donc possible de donner ce modèle en entré à MoBACT afin de générer sa configuration et les modèles de simulation qui serviront à le valider.

Nous illustrerons le reste de cette section avec les deux flux de données émis par les terminaux *controle1* et *controle2* à destination de *moteur1* et *moteur2*. Le système possède un total de 15 flux de données.

Le listing 9.9 présente les deux flux de données *consigne1* et *consigne2*. Ces deux flux de données appartiennent à la classe de trafic contrôle-commande et bénéficient donc de fenêtres protégées dans la configuration du TAS. Ils transmettent de nouvelles consignes aux terminaux *moteur1* et *moteur2* et leur chemin passe par les ponts *p1*, *p2* et *p3*.

Le listing 9.10 présente la configuration du TAS utilisée le long du chemin utilisé par les flux de données *consigne1* et *consigne2*. Ces deux flux de données ayant la même phase de

```
[...]
[stream2]
 name = "consigne_conn_controle1_consigne_moteur_emtr_pe_to_moteur1"
 payload.value = 38
 payload.unit = "B"
 type = "Periodic_Stream"
 period.value = 30000
 period.unit = "us"
 talker = ["::model::topologie2::controle1_to_p1.controle1_eth0"]
 listener = ["::model::topologie2::moteur1_to_p3.moteur1_eth0"]
 pcp = ["7"]
 offset.value = 11002
 offset.unit = "us"

[...]
[stream4]
 name = "consigne2_conn_controle2_consigne_moteur_emtr_pe_to_moteur2"
 payload.value = 38
 payload.unit = "B"
 type = "Periodic_Stream"
 period.value = 30000
 period.unit = "us"
 talker = ["::model::topologie2::controle2_to_p1.controle2_eth0"]
 listener = ["::model::topologie2::moteur2_to_p3.moteur2_eth0"]
 pcp = ["7"]
 offset.value = 11002
 offset.unit = "us"

[...]
```

Listing 9.9 – Paramètres des flux de données *consigne1* et *consigne2*.

transmission il est possible de regrouper la transmission de leurs trames dans une seule et même fenêtre protégée au niveau des ponts *p1* et *p2*.

Les figures 9.6 et 9.7 présentent les courbes de latence de bout en bout obtenues après la simulation du réseau avec NeSTiNg. Les valeurs de latence sont constantes et sont égales à 38,66 µs pour le premier et 47,76 µs pour le second, ils respectent dont leur échéance.

```
Switch: p1 (::model::topologie2::p1)
 [...]
 Port eth2: p1_eth2
  TAS configuration:
   Cycle duration: 30000us
   Time slot 0: duration: 11004us, priorities: [6, 5, 4, 3, 2, 1, 0]
   Time slot 1: duration: 6us, priorities: []
   Time slot 2: duration: 12us, priorities: [7]
   Time slot 3: duration: 18878us, priorities: [6, 5, 4, 3, 2, 1, 0]
 [...]

Switch: p2 (::model::topologie2::p2)
 [...]
 Port: eth3: p2_eth3
  TAS configuration:
   Cycle duration: 30000us
   Time slot 0: duration: 11018us, priorities: [6, 5, 4, 3, 2, 1, 0]
   Time slot 1: duration: 6us, priorities: []
   Time slot 2: duration: 12us, priorities: [7]
   Time slot 3: duration: 18864us, priorities: [6, 5, 4, 3, 2, 1, 0]
 [...]

Switch: p3 (::model::topologie2::p3)
 [...]
 Port: eth3: p3_eth1
  TAS configuration:
   Cycle duration: 30000us
   Time slot 0: duration: 11032us, priorities: [6, 5, 4, 3, 2, 1, 0]
   Time slot 1: duration: 6us, priorities: []
   Time slot 2: duration: 6us, priorities: [7]
   Time slot 3: duration: 18856us, priorities: [6, 5, 4, 3, 2, 1, 0]
 [...]
```

Listing 9.10 – Configuration du TAS le long du chemin emprunté par les flux de données *consigne1* et *consigne2*.

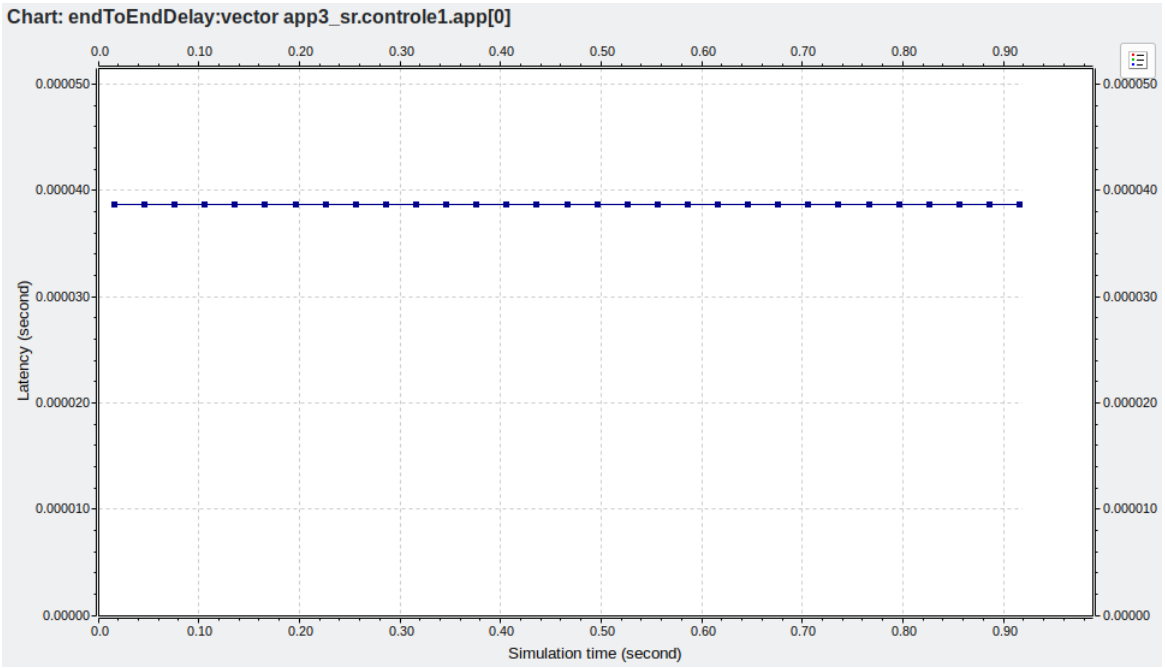


FIGURE 9.6 – Courbe de la latence de bout en bout du flux de données *consigne1* en utilisant le TAS avec NeSTiNg.

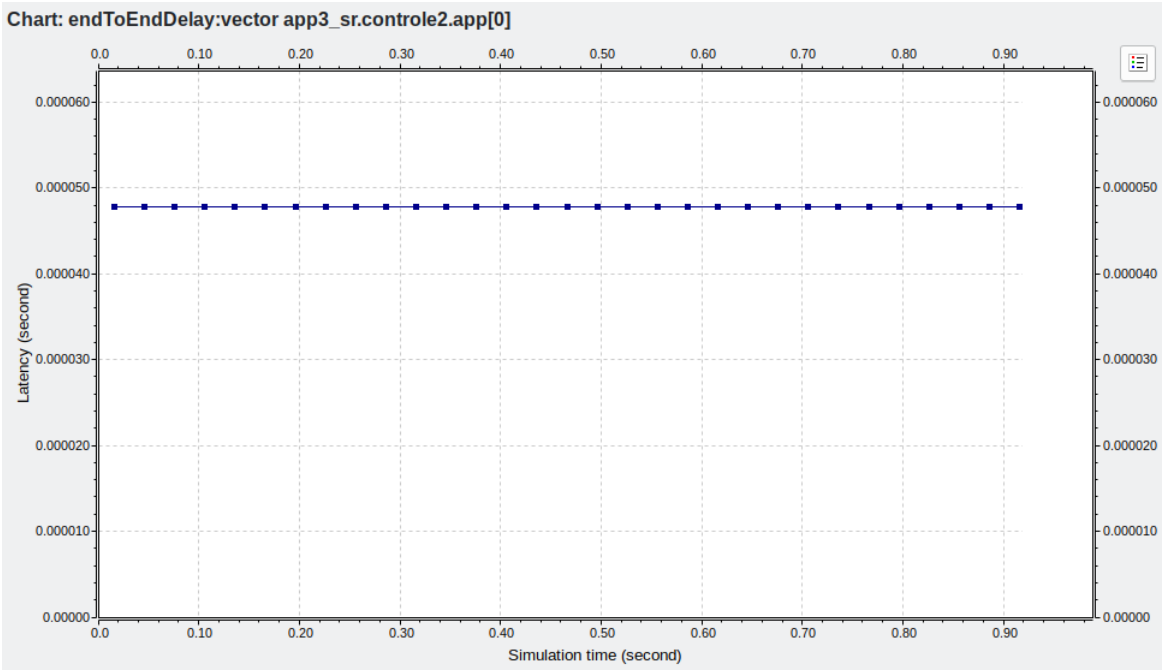


FIGURE 9.7 – Courbe de la latence de bout en bout du flux de données *consigne2* en utilisant le TAS avec NeSTiNg.

Conclusion et perspectives

Ce manuscrit de thèse a restitué les travaux réalisés dans le but de développer une approche de conception et de configuration de réseaux TSN basée sur la modélisation. Cette approche permet tout d'abord de maitriser la complexité du problème par la création d'un modèle dont les éléments représentent chaque élément du réseau. Nous considérons que la conception du système dans son entièreté, c'est-à-dire la partie logicielle et la partie réseau, est importante. Concevoir le système entièrement permet l'utilisation d'une telle approche en assurant la cohérence entre le comportement réel du système et le modèle qui est utilisé pour en produire la configuration. C'est pourquoi nous utilisons à la fois une approche de modélisation réseau et de modélisation logicielle.

Nous avons sélectionné le standard UCM pour la modélisation de l'architecture logicielle. L'outil Sigil-UCM nous permet de spécifier le comportement des composants formant l'application et le code technique, gérant notamment les communications réseau, de celle-ci. Notre approche s'appuie sur cette modélisation pour produire automatiquement le modèle des flux de données qui seront échangés sur le réseau en cours de conception. La génération du code et de ce modèle des flux de données à partir d'un unique modèle nous assure la cohérence entre les deux.

L'objectif de ces travaux était également de se servir de ces modèles pour produire une configuration du réseau, notamment de ses mécanismes d'ordonnancement. Ces mécanismes sont le moyen d'assurer le déterminisme du réseau et surtout de garantir le respect des exigences temps réel des flux de données. Enfin, une fois la configuration produite il est nécessaire de vérifier qu'elle répond correctement aux exigences du système. Cette étape est typiquement réalisée à l'aide de simulateurs réseau et d'outils d'analyse. Ces outils étant tous différents dans la façon de spécifier le problème à étudier, il est intéressant de générer automatiquement les modèles qu'ils utilisent dans le format qui leur est propre.

Le défi principal de ces travaux a été de mettre en place une approche complète et cohérente, permettant la formalisation du problème, la production de la configuration du réseau et la génération de modèles de simulation. Ces étapes ont du être pensées pour le domaine des système embarqués répartis et temps réel, dont la partie la plus critique des communications sont des flux de données de type contrôle-commande.

La première contribution de cette thèse est la proposition d'un ensemble de ressources de modélisation permettant de représenter l'ensemble des éléments d'un réseau TSN. Ces ressources de modélisation permettent de modéliser un réseau TSN de façon à pouvoir être utilisé par la suite pour générer automatiquement des modèles de simulation pour différents simulateurs réseau. Une syntaxe XML permettant d'exprimer ces modèles a également été définie afin de servir de format d'entrée pour un outil de génération de configuration et de modèles de simulation.

La deuxième contribution est la liaison de cette approche de modélisation réseau avec une approche de modélisation logicielle utilisant le standard UCM. Ce standard et l'outillage qui l'entoure, développé au sein du laboratoire de Thales Research & Technology dans lequel cette

thèse a été menée, permet la modélisation de l'architecture logicielle en la séparant en plusieurs composants. Il est ensuite possible de spécifier le comportement de ces composants, notamment des communications réseau qu'ils produiront. Nous avons développé un métamodèle permettant de regrouper cette spécification de comportement, des causes de déclenchement des composants et des liaisons qui existent entre eux. Grâce à ce métamodèle, nous avons développé un moyen de produire automatiquement le modèle des flux de données qui seront échangés sur le réseau dans le formalisme que nous avons défini précédemment.

Une fois la formalisation du problème effectuée, il est possible de produire la configuration des mécanismes d'ordonnancement du réseau TSN. Ces réseaux permettant la circulation de trafics de criticité mixte, plusieurs mécanismes d'ordonnancement peuvent être utilisés conjointement de façon à s'adapter aux besoins de chaque type de trafic. Néanmoins, l'utilisation de plusieurs mécanismes d'ordonnancement crée des interactions entre eux qui modifient la façon de calculer leur configuration afin de toujours respecter leur exigences temps réel. C'est le cas des deux mécanismes d'ordonnancement que nous utilisons dans nos travaux : le *Time Aware Shaper* (TAS) et le *Credit-Based Shaper* (CBS). Le calcul de la configuration du second est impacté par la configuration du premier. Notre troisième contribution est donc un moyen de calculer la configuration du CBS en prenant en considération la présence du TAS, ce qui augmente la quantité de ressource à allouer aux flux de données gérés par le CBS. Cette méthode de calcul définit des bornes inférieures pour la quantité de bande passante qu'il est nécessaire de réserver pour respecter les exigences temps réel.

La modélisation du réseau et la production de sa configuration permettent la génération de modèles de simulation et d'analyse permettant de vérifier leur validité. C'est l'objectif de l'outil que nous avons développé : MoBACT (*Model-Based Automatic Configuration for TSN*). MoBACT permet de faire appel a un outil externe pouvant synthétiser la configuration du TAS. Il permet ensuite de générer automatiquement des modèles de simulation et d'analyse pour différents outils. Ce processus de génération automatique permet de ne plus avoir besoin d'apprendre à se servir de chaque outil individuellement et surtout de ne pas devoir réaliser chaque modèle de simulation ou d'analyse manuellement, ce qui représente un important gain de temps. La production automatique de la configuration des mécanismes d'ordonnancement est indispensable pour éviter les erreurs et représente une autre économie de temps. L'utilisation de plusieurs outils permet la comparaison des résultats.

Enfin, nous présentons l'application de notre approche sur un système exemple. La mise en place d'une approche outillée telle que celle que nous proposons permet la mise en place d'un processus de conception itératif efficace. En effet, les changements successifs ne doivent être effectués qu'à un unique endroit : le modèle. Ces changements seront répercutés sur la production de la configuration et sur la génération des modèles de simulation et d'analyse. Nous présentons donc également l'application de notre approche sur des évolutions de ce système.

Nous avons identifié plusieurs perspectives de travaux futurs. La première d'entre elles est la réduction du pessimisme de notre approche de calcul de la configuration du CBS. En effet, l'approche que nous proposons est une approche locale, c'est-à-dire qu'elle permet de calculer la configuration du CBS à utiliser au niveau d'un port d'un pont et qu'il faut donc la répéter pour configurer l'ensemble du réseau.

Le calcul de la configuration est fait pour chaque port en tenant compte du pire cas qu'il est possible de rencontrer au niveau de ce port. La conséquence de cette façon de faire est que l'approche, lorsqu'elle est utilisée de façon répétée pour configurer le réseau, considère une somme de pire cas qui ne peut, en général, pas se produire. Le pire cas au niveau d'un port est le cas dans lequel la trame étudiée – pour laquelle nous effectuons le calcul de la configuration – est bloquée par le plus grand nombre d'autres trames possible. Le fait de répéter cette façon de

faire ne prend pas en considération le phénomène de sérialisation des trames. Les trames étant transmises les unes après les autres, les trames ayant bloqué la trame étudiée au niveau d'un port de sortie ne pourront pas toutes la bloquer à nouveau au niveau du port de sortie suivant.

Une perspective de travaux futurs permettant d'améliorer notre approche est donc la prise en considération de ce phénomène de sérialisation de trames. La réduction du pessimisme du calcul de la configuration du CBS permettrait de réduire le gâchis de ressources réseau causé par la réservation de bande passante pour un cas qui ne peut jamais se produire.

La deuxième perspective de travaux futurs est l'ajout à notre approche du support des flux de données *multicast*. Notre approche actuelle lie la modélisation réseau à la modélisation logicielle, faite grâce au standard UCM. L'outillage de ce standard que nous utilisons permet de générer le code technique – notamment responsable des communications réseau – et s'appuie sur une bibliothèque de communication ne supportant que l'utilisation de *sockets unicast*.

Cela ne concerne pas la partie modélisation réseau mais la possibilité de modéliser des flux de données *multicast* n'est pas proposée pour ne pas permettre la modélisation d'un système dont le code ne peut pas être généré. Les ressources de modélisation réseau ont été conçues originellement pour supporter les flux *multicast*, et ajouter cette possibilité ne représente pas de défi particulier. En revanche, afin de pouvoir continuer à utiliser notre approche dans son entièreté, il est nécessaire d'utiliser une bibliothèque de communication supportant les flux de données *multicast*.

La troisième perspective importante de travaux futurs concerne la génération de fichier de configuration pour du matériel réseau. La version actuelle de MoBACT permet la génération de tels fichiers pour un outil de configuration de pont TSN spécifique au modèle que nous utilisons dans notre laboratoire. Une évolution possible de MoBACT serait l'implémentation d'un générateur de fichier de configuration de matériel dans un format général, faisant partie des standards TSN et devant donc être supporté par l'ensemble des équipements réseau supportant les standards TSN.

Ce format utilise les modèles de données YANG [M.10] qui contiennent la configuration à déployer sur les équipement réseau par le biais du protocole NETCONF [RMJA11]. Générer des fichiers de configuration dans ce format général permettrait de pouvoir facilement déployer la configuration du réseau produite par MoBACT sur n'importe quel équipement réseau implémentant le support de ce format, en accord avec les standards TSN.

L'utilisation combinée dans notre approche de la modélisation réseau et de la modélisation logicielle implique que notre approche vise à concevoir des systèmes dans leur intégralité. C'est un parti pris qui nous permet de nous assurer que la configuration qui sera générée correspondra exactement aux besoins des applications qui utiliseront le réseau en garantissant la cohérence entre leur comportement réel, de part la génération automatique de leur code, et le modèle utilisé pour générer la configuration, de part la modélisation automatique des flux de données. Néanmoins, une approche complète de ce type n'est pas forcément adaptée à tous les contextes industriels. En effet, certains domaines n'ont pas pour habitude de concevoir entièrement leurs systèmes mais plutôt d'utiliser des composants sur étagères [22] (calculateurs et autres terminaux) et donc de ne pas réaliser la conception des applications.

Nous avons constaté l'efficacité de MoBACT pour la mise en place de démonstrations. Nous avons pu l'utiliser pour mettre en place des démonstrations de conception de réseau TSN rapidement. Lors de ces mises en œuvre de MoBACT, la création du modèle prenait environ une journée et la génération de la configuration et des modèles de simulation et d'analyse quelques minutes alors que le faire sans assistance aurait pris plusieurs jours.

22. Des composants COTS, pour *Commercial off-the-shelf* en anglais.

Nous avons également pu constater les possibilités d'évolution de cet outil. Un prototype de solution de génération du configuration pour le TAS a été développé au sein du laboratoire utilisant une méthode et un solveur différent de celui auquel fait originellement appel MoBACT. L'ajout de nouvelles cibles vers lesquelles générer des modèles de simulation a également pu être testée et s'est avérée particulièrement aisée.

Les différentes contributions présentées dans cette thèse sont indépendantes les unes des autres, il n'est pas obligatoire de toutes les utilisées ensemble pour en tirer parti. La syntaxe que nous proposons pour la modélisation du réseau peut être utilisée par d'autres outils que MoBACT. Nous utilisons le standard UCM pour modéliser l'application qui utilisera le réseau mais il est possible d'utiliser n'importe quel autre formalisme si une traduction vers le métamodèle *verif_exec* est possible. La modélisation automatique des flux de données peut être utile même dans des cas où TSN n'est pas utilisé et ou aucune configuration n'est à calculer, à de simples fins de documentation par exemple. Si un réseau utilisé dans un système déjà existant doit évolué pour utiliser les fonctionnalités de TSN et que ses flux de données sont déjà connus et caractérisés, il est possible d'utiliser MoBACT sans passer par les étapes précédentes de modélisation.

Annexe A

Ressources de modélisation

A.1 Ressources de modélisation au format XML

A.1.1 Modèle des ressources de modélisation pour réseaux Ethernet

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<resourceModule name="ethernet_rsc">
 <contractModule name="types">
  <integer kind="uint32" name="bandwidth_value_t"/>
  <enum indexType="uint16" name="bandwidth_unit_t">
   <value index="0" name="kbps"/>
   <value index="1" name="Mbps"/>
   <value index="2" name="Gbps"/>
  </enum>
  <struct name="bandwidth_t">
   <field name="bandwidth_value" type="bandwidth_value_t"/>
   <field name="bandwidth_unit" type="bandwidth_unit_t"/>
  </struct>
  <string base="wchar" maxSize="0" name="MAC_address_t"/>
  <string base="wchar" maxSize="0" name="IP_adress_t"/>
  <string base="wchar" maxSize="0" name="netmask_t"/>
  <struct name="IP_t">
   <field name="IP_adress" type="IP_adress_t"/>
   <field name="netmask" type="netmask_t"/>
  </struct>
 </contractModule>
```

```
22 </resourceModule name="ethernet_rsc">
23  [...]
24  <commRscDef name="Ethernet_Link">
25   <commRscPort def="Ethernet_interface" max="2" min="2" name="
      ↪ ethernet_interfaces_el"/>
26   <configParam name="propagation_delay" type="::tsn_rsc::types::time_t"/
      ↪ >
27  </commRscDef>
28  <commPortDef name="Ethernet_interface">
29   <configParam name="bandwidth" type="types::bandwidth_t"/>
30  </commPortDef>
31  <commPortDef name="Eth_IP_interface">
32   <extends ref="Ethernet_interface"/>
33   <configParam name="IP_address" type="types::IP_t"/>
34   <configParam name="MAC_address" type="types::MAC_address_t"/>
35  </commPortDef>
36  <computRscDef name="End_Point">
37   <computRscPort def="Eth_IP_interface" max="1" min="0" name="ep_eth0"/>
38   <domainParam max="-1" min="0" name="deployment_specification"/>
39  </computRscDef>
40  <computRscDef name="End_Point_2_Ports">
41   <extends ref="End_Point"/>
42   <computRscPort def="Eth_IP_interface" max="1" min="0" name="ep_eth1"/>
43  </computRscDef>
44  <computRscDef name="End_Point_3_Ports">
45   <extends ref="End_Point_2_Ports"/>
46   <computRscPort def="Eth_IP_interface" max="1" min="0" name="ep_eth2"/>
47  </computRscDef>
48  <computRscDef name="End_Point_4_Ports">
49   <extends ref="End_Point_3_Ports"/>
50   <computRscPort def="Eth_IP_interface" max="1" min="0" name="ep_eth3"/>
51  </computRscDef>
52  <computRscDef name="Ethernet_Switch">
53   <computRscPort def="Ethernet_interface" max="2" min="-1" name="es_ethX
      ↪ "/>
54  </computRscDef>
55 </resourceModule>
```

Listing A.11 – Modèle de ressources contenant la définition des concepts servant à modéliser des réseaux Ethernet au format XML utilisé par Sigil-UCM.

A.1.2 Modèle des ressources de modélisation pour réseaux TSN

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<resourceModule name="tsn_rsc">
 <contractModule name="types">
  <enum indexType="uint8" name="TSN_Functionalities">
   <value index="0" name="802_1Qav"/>
   <value index="1" name="802_1Qbv"/>
   <value index="2" name="802_1Qbu"/>
   <value index="3" name="802_1AS"/>
   <value index="4" name="802_1CB"/>
  </enum>
  <sequence indexType="uint32" name="supported_functionalities" type="
   ↪ TSN_Functionalities"/>
  <float kind="float" name="bandwidth_proportion_t"/>
  <struct name="idle_slope_t">
   <field name="bandwidth_proportion" type="bandwidth_proportion_t"/>
   <field name="pcp" type="pcp_t"/>
  </struct>
  <integer kind="uint32" name="time_value_t"/>
  <enum indexType="uint16" name="time_unit_t">
   <value index="0" name="s"/>
   <value index="1" name="ms"/>
   <value index="2" name="us"/>
   <value index="3" name="ns"/>
  </enum>
  <struct name="time_t">
   <field name="time_value" type="time_value_t"/>
   <field name="time_unit" type="time_unit_t"/>
  </struct>
  <sequence indexType="uint8" maxSize="8" name="pcp_seq_t" type="pcp_t"/
   ↪ >
  <struct name="time_slot_t">
   <field name="duration" type="time_t"/>
   <field name="allowed_priorities" type="pcp_seq_t"/>
  </struct>
  <enum indexType="uint16" name="data_unit_t">
   <value index="0" name="b"/>
   <value index="1" name="B"/>
   <value index="2" name="kb"/>
   <value index="3" name="kB"/>
   <value index="4" name="Mb"/>
   <value index="5" name="MB"/>
   <value index="6" name="Gb"/>
   <value index="7" name="GB"/>
  </enum>
```

```
43    <integer kind="uint32" name="data_size_t"/>
44    <struct name="payload_t">
45     <field name="data_size" type="data_size_t"/>
46     <field name="data_unit" type="data_unit_t"/>
47    </struct>
48    <integer kind="uint8" name="vlan_id_t"/>
49    <bool name="is_critical_t"/>
50    <integer kind="uint8" name="pcp_t"/>
51   </contractModule>
52   <commRscDef name="Applicative_Stream">
53    <configParam name="payload" type="types::payload_t"/>
54    <configParam name="pcp" type="types::pcp_t"/>
55    <configParam min="0" name="offset" type="types::time_t"/>
56    <configParam min="0" name="end_time" type="types::time_t"/>
57    <rscParam max="1" min="0" name="listener">
58     <allowedRscDef ref="::ethernet_rsc::Eth_IP_interface"/>
59    </rscParam>
60    <rscParam max="1" min="0" name="talker">
61     <allowedRscDef ref="::ethernet_rsc::Eth_IP_interface"/>
62    </rscParam>
63   </commRscDef>
64   <commRscDef name="Periodic_Stream">
65    <extends ref="Applicative_Stream"/>
66    <configParam name="period" type="types::time_t"/>
67   </commRscDef>
68   <commRscDef name="Sporadic_Stream">
69    <extends ref="Applicative_Stream"/>
70    <configParam name="min_interval" type="types::time_t"/>
71   </commRscDef>
72   <commRscDef name="Aperiodic_Stream">
73    <extends ref="Applicative_Stream"/>
74   </commRscDef>
75   <commRscDef name="Stream_System_Requirements">
76    <rscParam max="-1" min="1" name="Stream">
77     <allowedRscDef ref="Applicative_Stream"/>
78    </rscParam>
79    <configParam min="0" name="deadline" type="types::time_t"/>
80    <configParam name="vlan_id" type="types::vlan_id_t"/>
81    <configParam min="0" name="maximum_jitter" type="types::time_t"/>
```

```
82    <configParam defaultValue="false" name="is_critical" type="types::
       ↪ is_critical_t"/>
83    <rscParam max="1" min="0" name="overridden_listener">
84     <allowedRscDef ref="::ethernet_rsc::Eth_IP_interface"/>
85    </rscParam>
86    <rscParam max="1" min="0" name="overridden_talker">
87     <allowedRscDef ref="::ethernet_rsc::Eth_IP_interface"/>
88    </rscParam>
89    <structParam max="-1" min="1" name="path">
90     <rscParam max="-1" min="0" name="path_member">
91      <allowedRscDef ref="::ethernet_rsc::Ethernet_interface"/>
92     </rscParam>
93    </structParam>
94   </commRscDef>
95   <computRscDef name="TSN_Switch">
96    <computRscPort def="::ethernet_rsc::Ethernet_interface" max="1" min="0
       ↪ " name="ts_eth0"/>
97    <computRscPort def="::ethernet_rsc::Ethernet_interface" max="1" min="0
       ↪ " name="ts_eth1"/>
98    <computRscPort def="::ethernet_rsc::Ethernet_interface" max="1" min="0
       ↪ " name="ts_eth2"/>
99    <configParam max="-1" min="0" name="supported_functionalities" type="
       ↪ types::supported_functionalities"/>
100   <configParam name="processing_delay" type="types::time_t"/>
101  </computRscDef>
102  <computRscDef name="TSN_Switch_4_Ports">
103   <extends ref="TSN_Switch"/>
104   <computRscPort def="::ethernet_rsc::Ethernet_interface" max="1" min="0
       ↪ " name="ts_eth3"/>
105  </computRscDef>
106  <computRscDef name="TSN_Switch_6_Ports">
107   <extends ref="TSN_Switch_4_Ports"/>
108   <computRscPort def="::ethernet_rsc::Ethernet_interface" max="1" min="0
       ↪ " name="ts_eth4"/>
109   <computRscPort def="::ethernet_rsc::Ethernet_interface" max="1" min="0
       ↪ " name="ts_eth5"/>
110  </computRscDef>
111  <computRscDef name="TSN_Switch_8_Ports">
112   <extends ref="TSN_Switch_6_Ports"/>
113   <computRscPort def="::ethernet_rsc::Ethernet_interface" max="1" min="0
       ↪ " name="ts_eth6"/>
114   <computRscPort def="::ethernet_rsc::Ethernet_interface" max="1" min="0
       ↪ " name="ts_eth7"/>
115  </computRscDef>
116 </resourceModule>
```

Listing A.12 – Modèle de ressources contenant la définition des concepts utilisés pour modéliser des réseaux TSN au format XML utilisé par Sigil-UCM.

A.2 Modèle du réseau servant d'exemple

A.2.1 Modèle de la topologie du réseau servant d'exemple

```
<environment name="topologie1">
  <computRsc def="::ethernet_rsc::End_Point" name="controle">
   <computPortConf>
    <computPortRef ref="ep_eth0"/>
    <config def="IP_address" value="{&quot;192.168.0.1&quot;,&quot;/24&
   ↪ quot;}"/>
   </computPortConf>
  </computRsc>
  <computRsc def="::ethernet_rsc::End_Point" name="moteur">
   <computPortConf>
    <computPortRef ref="ep_eth0"/>
    <config def="IP_address" value="{&quot;192.168.0.2&quot;,&quot;/24&
   ↪ quot;}"/>
   </computPortConf>
  </computRsc>
  <computRsc def="::ethernet_rsc::End_Point" name="mission">
   <computPortConf>
    <computPortRef ref="ep_eth0"/>
    <config def="IP_address" value="{&quot;192.168.0.3&quot;,&quot;/24&
   ↪ quot;}"/>
   </computPortConf>
  </computRsc>
  <computRsc def="::ethernet_rsc::End_Point" name="passerelle">
   <computPortConf>
    <computPortRef ref="ep_eth0"/>
    <config def="IP_address" value="{&quot;192.168.0.4&quot;,&quot;/24&
   ↪ quot;}"/>
   </computPortConf>
  </computRsc>
  <computRsc def="::tsn_rsc::TSN_Switch_4_Ports" name="pont1">
   <computPortConf>
    <computPortRef ref="ts_eth0"/>
    <computPortRef ref="ts_eth1"/>
    <computPortRef ref="ts_eth2"/>
    <computPortRef ref="ts_eth3"/>
   </computPortConf>
  </computRsc>
```

```
34    <computRsc def="::tsn_rsc::TSN_Switch_4_Ports" name="pont2">
35     <computPortConf>
36      <computPortRef ref="ts_eth0"/>
37      <computPortRef ref="ts_eth1"/>
38      <computPortRef ref="ts_eth2"/>
39      <computPortRef ref="ts_eth3"/>
40     </computPortConf>
41    </computRsc>
42    <commRsc def="::ethernet_rsc::Ethernet_Link" name="passerelle_to_pont1
       ↪ ">
43     <commPort computRsc="passerelle" computRscPort="ep_eth0" name="
       ↪ passerelle_eth0"/>
44     <commPort computRsc="pont1" computRscPort="ts_eth0" name="pont1_et0"/
       ↪ >
45    </commRsc>
46    <commRsc def="::ethernet_rsc::Ethernet_Link" name="moteur_to_pont1">
47     <commPort computRsc="moteur" computRscPort="ep_eth0" name="
       ↪ moteur_eth0"/>
48     <commPort computRsc="pont1" computRscPort="ts_eth1" name="pont1_eth1"
       ↪ />
49    </commRsc>
50    <commRsc def="::ethernet_rsc::Ethernet_Link" name="controle_to_pont2">
51     <commPort computRsc="controle" computRscPort="ep_eth0" name="
       ↪ controle_eth0"/>
52     <commPort computRsc="pont2" computRscPort="ts_eth0" name="pont2_eth0"
       ↪ />
53    </commRsc>
54    <commRsc def="::ethernet_rsc::Ethernet_Link" name="mission_to_pont2">
55     <commPort computRsc="mission" computRscPort="ep_eth0" name="
       ↪ mission_eth0"/>
56     <commPort computRsc="pont2" computRscPort="ts_eth1" name="pont2_eth1"
       ↪ />
57    </commRsc>
58    <commRsc def="::ethernet_rsc::Ethernet_Link" name="pont1_to_pont2">
59     <commPort computRsc="pont1" computRscPort="ts_eth2" name="pont1_eth2"
       ↪ />
60     <commPort computRsc="pont2" computRscPort="ts_eth2" name="pont2_eth2"
       ↪ />
61    </commRsc>
62  </environment>
```

Listing A.13 – Modèle du réseau utilisé comme exemple contenant l'instanciation de la topologie et des exigences système.

A.2.2 Modèle des flux de données du réseau servant d'exemple

```
<environment name="streams">
 <commRsc def="::tsn_rsc::Periodic_Stream" name="
   ↪ conn_passerelle_passerelle1_ordre_emtr_pe_to_mission1">
  <comment>stream1</comment>
  <config def="pcp" value="5"/>
  <config def="payload" value="{60, B}"/>
  <config def="offset" value="{2, us}"/>
  <config def="period" value="{180000, us}"/>
  <rscConfig def="talker" value="::model::topologie1::
   ↪ passerelle_to_pont1.passerelle_eth0"/>
  <rscConfig def="listener" value="::model::topologie1::mission_to_pont2
   ↪ .mission_eth0"/>
 </commRsc>
 <commRsc def="::tsn_rsc::Periodic_Stream" name="
   ↪ consigne_conn_controle1_consigne_moteur_emtr_pe_to_moteur1">
  <comment>stream2</comment>
  <config def="pcp" value="7"/>
  <config def="payload" value="{38, B}"/>
  <config def="offset" value="{12004, us}"/>
  <config def="period" value="{30000, us}"/>
  <rscConfig def="talker" value="::model::topologie1::controle_to_pont2.
   ↪ controle_eth0"/>
  <rscConfig def="listener" value="::model::topologie1::moteur_to_pont1.
   ↪ moteur_eth0"/>
 </commRsc>
 <commRsc def="::tsn_rsc::Periodic_Stream" name="
   ↪ etat_conn_moteur1_etat_emtr_pe_to_controle1">
  <comment>stream3</comment>
  <config def="pcp" value="7"/>
  <config def="payload" value="{44, B}"/>
  <config def="offset" value="{22004, us}"/>
  <config def="period" value="{30000, us}"/>
  <rscConfig def="talker" value="::model::topologie1::moteur_to_pont1.
   ↪ moteur_eth0"/>
  <rscConfig def="listener" value="::model::topologie1::
   ↪ controle_to_pont2.controle_eth0"/>
 </commRsc>
```

```
29  <commRsc def="::tsn_rsc::Periodic_Stream" name="
      ↪ ordre_mission_mission1_ordre_emtr_pe_to_controle1">
30   <comment>stream4</comment>
31   <config def="pcp" value="6"/>
32   <config def="payload" value="{60, B}"/>
33   <config def="offset" value="{10002, us}"/>
34   <config def="period" value="{30000, us}"/>
35   <rscConfig def="talker" value="::model::topologie1::mission_to_pont2.
      ↪ mission_eth0"/>
36   <rscConfig def="listener" value="::model::topologie1::
      ↪ controle_to_pont2.controle_eth0"/>
37  </commRsc>
38 </environment>
```

Listing A.14 – Modèle du réseau utilisé comme exemple contenant l'instanciation des flux de données.

A.2.3 Modèle des exigences système du réseau servant d'exemple

```
<environment name="app1_sr">
 <commRsc def="::tsn_rsc::Stream_System_Requirements" name="stream1">
  <rscConfig def="Stream" value="::streams::
   ↪ conn_passerelle_passerelle1_ordre_emtr_pe_to_mission1"/>
  <rscConfig def="overridden_talker" value="::model::topologie1::
   ↪ passerelle_to_pont1.passerelle_eth0"/>
  <rscConfig def="overridden_listener" value="::model::topologie1::
   ↪ mission_to_pont2.mission_eth0"/>
  <structConfig def="path">
   <rscConfig def="path_member" value="::model::topologie1::
   ↪ moteur_to_pont1.pont1_eth1"/>
   <rscConfig def="path_member" value="::model::topologie1::
   ↪ pont1_to_pont2.pont2_eth2"/>
  </structConfig>
  <config def="vlan_id" value="1"/>
  <config def="deadline" value="{100,ms}"/>
  <config def="is_critical" value="false"/>
 </commRsc>
 <commRsc def="::tsn_rsc::Stream_System_Requirements" name="stream2">
  <rscConfig def="Stream" value="::streams::
   ↪ consigne_conn_controle1_consigne_moteur_emtr_pe_to_moteur1"/>
  <rscConfig def="overridden_talker" value="::model::topologie1::
   ↪ controle_to_pont2.controle_eth0"/>
  <rscConfig def="overridden_listener" value="::model::topologie1::
   ↪ moteur_to_pont1.moteur_eth0"/>
  <structConfig def="path">
   <rscConfig def="path_member" value="::model::topologie1::
   ↪ controle_to_pont2.pont2_eth0"/>
   <rscConfig def="path_member" value="::model::topologie1::
   ↪ pont1_to_pont2.pont1_eth2"/>
  </structConfig>
  <config def="vlan_id" value="1"/>
  <config def="deadline" value="{10,ms}"/>
  <config def="maximum_jitter" value="{1,ms}"/>
  <config def="is_critical" value="true"/>
 </commRsc>
```

```
27   <commRsc def="::tsn_rsc::Stream_System_Requirements" name="stream3">
28    <rscConfig def="Stream" value="::streams::
       ↪ etat_conn_moteur1_etat_emtr_pe_to_controle1"/>
29    <rscConfig def="overridden_talker" value="::model::topologie1::
       ↪ moteur_to_pont1.moteur_eth0"/>
30    <rscConfig def="overridden_listener" value="::model::topologie1::
       ↪ controle_to_pont2.controle_eth0"/>
31    <structConfig def="path">
32     <rscConfig def="path_member" value="::model::topologie1::
       ↪ moteur_to_pont1.pont1_eth1"/>
33     <rscConfig def="path_member" value="::model::topologie1::
       ↪ pont1_to_pont2.pont2_eth2"/>
34    </structConfig>
35    <config def="vlan_id" value="1"/>
36    <config def="deadline" value="{10,ms}"/>
37    <config def="maximum_jitter" value="{1,ms}"/>
38    <config def="is_critical" value="true"/>
39   </commRsc>
40   <commRsc def="::tsn_rsc::Stream_System_Requirements" name="stream4">
41    <rscConfig def="Stream" value="::streams::
       ↪ ordre_mission_mission1_ordre_emtr_pe_to_controle1"/>
42    <rscConfig def="overridden_talker" value="::model::topologie1::
       ↪ mission_to_pont2.mission_eth0"/>
43    <rscConfig def="overridden_listener" value="::model::topologie1::
       ↪ controle_to_pont2.controle_eth0"/>
44    <structConfig def="path">
45     <rscConfig def="path_member" value="::model::topologie1::
       ↪ mission_to_pont2.pont2_eth1"/>
46    </structConfig>
47    <config def="vlan_id" value="1"/>
48    <config def="deadline" value="{100,ms}"/>
49    <config def="is_critical" value="false"/>
50   </commRsc>
51 </environment>
```

Listing A.15 – Modèle du réseau utilisé comme exemple contenant les exigences système.

A.3 Modèle d'un réseau plus complexe

A.3.1 Modèle de la topologie d'un réseau plus complexe

```
<environment name="topologie2">
 <computRsc def="::ethernet_rsc::End_Point" name="controle1">
  <computPortConf>
   <computPortRef ref="ep_eth0"/>
   <config def="IP_address" value="{&quot;192.168.0.1&quot;,&quot;/24&
   ↪ quot;}"/>
  </computPortConf>
 </computRsc>
 <computRsc def="::ethernet_rsc::End_Point" name="controle2">
  <computPortConf>
   <computPortRef ref="ep_eth0"/>
   <config def="IP_address" value="{&quot;192.168.0.2&quot;,&quot;/24&
   ↪ quot;}"/>
  </computPortConf>
 </computRsc>
 <computRsc def="::ethernet_rsc::End_Point" name="controle3">
  <computPortConf>
   <computPortRef ref="ep_eth0"/>
   <config def="IP_address" value="{&quot;192.168.0.3&quot;,&quot;/24&
   ↪ quot;}"/>
  </computPortConf>
 </computRsc>
 <computRsc def="::ethernet_rsc::End_Point" name="controle4">
  <computPortConf>
   <computPortRef ref="ep_eth0"/>
   <config def="IP_address" value="{&quot;192.168.0.4&quot;,&quot;/24&
   ↪ quot;}"/>
  </computPortConf>
 </computRsc>
 <computRsc def="::ethernet_rsc::End_Point" name="moteur1">
  <computPortConf>
   <computPortRef ref="ep_eth0"/>
   <config def="IP_address" value="{&quot;192.168.0.5&quot;,&quot;/24&
   ↪ quot;}"/>
  </computPortConf>
 </computRsc>
```

```
32  <computRsc def="::ethernet_rsc::End_Point" name="moteur2">
33   <computPortConf>
34    <computPortRef ref="ep_eth0"/>
35    <config def="IP_address" value="{&quot;192.168.0.6&quot;,&quot;/24&
      ↪ quot;}"/>
36   </computPortConf>
37  </computRsc>
38  <computRsc def="::ethernet_rsc::End_Point" name="moteur3">
39   <computPortConf>
40    <computPortRef ref="ep_eth0"/>
41    <config def="IP_address" value="{&quot;192.168.0.7&quot;,&quot;/24&
      ↪ quot;}"/>
42   </computPortConf>
43  </computRsc>
44  <computRsc def="::ethernet_rsc::End_Point" name="moteur4">
45   <computPortConf>
46    <computPortRef ref="ep_eth0"/>
47    <config def="IP_address" value="{&quot;192.168.0.8&quot;,&quot;/24&
      ↪ quot;}"/>
48   </computPortConf>
49  </computRsc>
50  <computRsc def="::ethernet_rsc::End_Point" name="boussole">
51   <computPortConf>
52    <computPortRef ref="ep_eth0"/>
53    <config def="IP_address" value="{&quot;192.168.0.9&quot;,&quot;/24&
      ↪ quot;}"/>
54   </computPortConf>
55  </computRsc>
56  <computRsc def="::ethernet_rsc::End_Point" name="position">
57   <computPortConf>
58    <computPortRef ref="ep_eth0"/>
59    <config def="IP_address" value="{&quot;192.168.0.10&quot;,&quot;/24&
      ↪ quot;}"/>
60   </computPortConf>
61  </computRsc>
62  <computRsc def="::ethernet_rsc::End_Point" name="mission1">
63   <computPortConf>
64    <computPortRef ref="ep_eth0"/>
65    <config def="IP_address" value="{&quot;192.168.0.11&quot;,&quot;/24&
      ↪ quot;}"/>
66   </computPortConf>
67  </computRsc>
```

```
68  <computRsc def="::ethernet_rsc::End_Point" name="passerelle1">
69   <computPortConf>
70    <computPortRef ref="ep_eth0"/>
71    <config def="IP_address" value="{&quot;192.168.0.12&quot;,&quot;/24&
      ↪ quot;}"/>
72   </computPortConf>
73  </computRsc>
74  <computRsc def="::tsn_rsc::TSN_Switch_4_Ports" name="p1">
75   <computPortConf>
76    <computPortRef ref="ts_eth0"/>
77    <computPortRef ref="ts_eth1"/>
78    <computPortRef ref="ts_eth2"/>
79    <computPortRef ref="ts_eth3"/>
80   </computPortConf>
81   <config def="processing_delay" value="{2, us}"/>
82  </computRsc>
83  <computRsc def="::tsn_rsc::TSN_Switch_4_Ports" name="p2">
84   <computPortConf>
85    <computPortRef ref="ts_eth0"/>
86    <computPortRef ref="ts_eth1"/>
87    <computPortRef ref="ts_eth2"/>
88    <computPortRef ref="ts_eth3"/>
89   </computPortConf>
90   <config def="processing_delay" value="{2, us}"/>
91  </computRsc>
92  <computRsc def="::tsn_rsc::TSN_Switch_4_Ports" name="p3">
93   <computPortConf>
94    <computPortRef ref="ts_eth0"/>
95    <computPortRef ref="ts_eth1"/>
96    <computPortRef ref="ts_eth2"/>
97    <computPortRef ref="ts_eth3"/>
98   </computPortConf>
99   <config def="processing_delay" value="{2, us}"/>
100 </computRsc>
```

```
101  <computRsc def="::tsn_rsc::TSN_Switch_4_Ports" name="p4">
102   <computPortConf>
103    <computPortRef ref="ts_eth0"/>
104    <computPortRef ref="ts_eth1"/>
105    <computPortRef ref="ts_eth2"/>
106    <computPortRef ref="ts_eth3"/>
107   </computPortConf>
108   <config def="processing_delay" value="{2, us}"/>
109  </computRsc>
110  <computRsc def="::tsn_rsc::TSN_Switch_4_Ports" name="p5">
111   <computPortConf>
112    <computPortRef ref="ts_eth0"/>
113    <computPortRef ref="ts_eth1"/>
114    <computPortRef ref="ts_eth2"/>
115    <computPortRef ref="ts_eth3"/>
116   </computPortConf>
117   <config def="processing_delay" value="{2, us}"/>
118  </computRsc>
119  <computRsc def="::tsn_rsc::TSN_Switch_4_Ports" name="p6">
120   <computPortConf>
121    <computPortRef ref="ts_eth0"/>
122    <computPortRef ref="ts_eth1"/>
123    <computPortRef ref="ts_eth2"/>
124    <computPortRef ref="ts_eth3"/>
125   </computPortConf>
126   <config def="processing_delay" value="{2, us}"/>
127  </computRsc>
128  <commRsc def="::ethernet_rsc::Ethernet_Link" name="controle1_to_p1">
129   <commPort computRsc="controle1" computRscPort="ep_eth0" name="
      ↪ controle1_eth0"/>
130   <commPort computRsc="p1" computRscPort="ts_eth0" name="p1_eth0"/>
131  </commRsc>
132  <commRsc def="::ethernet_rsc::Ethernet_Link" name="controle2_to_p1">
133   <commPort computRsc="controle2" computRscPort="ep_eth0" name="
      ↪ controle2_eth0"/>
134   <commPort computRsc="p1" computRscPort="ts_eth1" name="p1_eth1"/>
135  </commRsc>
```

```
136  <commRsc def="::ethernet_rsc::Ethernet_Link" name="boussole_to_p2">
137   <commPort computRsc="boussole" computRscPort="ep_eth0" name="
       ↪ boussole_eth0"/>
138   <commPort computRsc="p2" computRscPort="ts_eth1" name="p2_eth1"/>
139  </commRsc>
140  <commRsc def="::ethernet_rsc::Ethernet_Link" name="position_to_p2">
141   <commPort computRsc="position" computRscPort="ep_eth0" name="
       ↪ position_eth0"/>
142   <commPort computRsc="p2" computRscPort="ts_eth2" name="p2_eth2"/>
143  </commRsc>
144  <commRsc def="::ethernet_rsc::Ethernet_Link" name="p2_to_p3">
145   <commPort computRsc="p2" computRscPort="ts_eth3" name="p2_eth3"/>
146   <commPort computRsc="p3" computRscPort="ts_eth0" name="p3_eth0"/>
147  </commRsc>
148  <commRsc def="::ethernet_rsc::Ethernet_Link" name="moteur1_to_p3">
149   <commPort computRsc="moteur1" computRscPort="ep_eth0" name="
       ↪ moteur1_eth0"/>
150   <commPort computRsc="p3" computRscPort="ts_eth1" name="p3_eth1"/>
151  </commRsc>
152  <commRsc def="::ethernet_rsc::Ethernet_Link" name="moteur2_to_p3">
153   <commPort computRsc="moteur2" computRscPort="ep_eth0" name="
       ↪ moteur2_eth0"/>
154   <commPort computRsc="p3" computRscPort="ts_eth2" name="p3_eth2"/>
155  </commRsc>
156  <commRsc def="::ethernet_rsc::Ethernet_Link" name="p3_to_p4">
157   <commPort computRsc="p3" computRscPort="ts_eth3" name="p3_eth3"/>
158   <commPort computRsc="p4" computRscPort="ts_eth0" name="p4_eth0"/>
159  </commRsc>
160  <commRsc def="::ethernet_rsc::Ethernet_Link" name="moteur3_to_p4">
161   <commPort computRsc="moteur3" computRscPort="ep_eth0" name="
       ↪ moteur3_eth0"/>
162   <commPort computRsc="p4" computRscPort="ts_eth1" name="p4_eth1"/>
163  </commRsc>
164  <commRsc def="::ethernet_rsc::Ethernet_Link" name="moteur4_to_p4">
165   <commPort computRsc="moteur4" computRscPort="ep_eth0" name="
       ↪ moteur4_eth0"/>
166   <commPort computRsc="p4" computRscPort="ts_eth2" name="p4_eth2"/>
167  </commRsc>
```

```
168  <commRsc def="::ethernet_rsc::Ethernet_Link" name="p1_to_p2">
169   <commPort computRsc="p1" computRscPort="ts_eth2" name="p1_eth2"/>
170   <commPort computRsc="p2" computRscPort="ts_eth0" name="p2_eth0"/>
171  </commRsc>
172  <commRsc def="::ethernet_rsc::Ethernet_Link" name="p4_to_p5">
173   <commPort computRsc="p4" computRscPort="ts_eth3" name="p4_eth3"/>
174   <commPort computRsc="p5" computRscPort="ts_eth0" name="p5_eth0"/>
175  </commRsc>
176  <commRsc def="::ethernet_rsc::Ethernet_Link" name="mission1_to_p5">
177   <commPort computRsc="mission1" computRscPort="ep_eth0" name="
      ↪ mission1_eth0"/>
178   <commPort computRsc="p5" computRscPort="ts_eth1" name="p5_eth1"/>
179  </commRsc>
180  <commRsc def="::ethernet_rsc::Ethernet_Link" name="passerelle1_to_p5">
181   <commPort computRsc="passerelle1" computRscPort="ep_eth0" name="
      ↪ passerelle1_eth0"/>
182   <commPort computRsc="p5" computRscPort="ts_eth2" name="p5_eth2"/>
183  </commRsc>
184  <commRsc def="::ethernet_rsc::Ethernet_Link" name="p5_to_p6">
185   <commPort computRsc="p5" computRscPort="ts_eth3" name="p5_eth3"/>
186   <commPort computRsc="p6" computRscPort="ts_eth0" name="p6_eth0"/>
187  </commRsc>
188  <commRsc def="::ethernet_rsc::Ethernet_Link" name="controle3_to_p6">
189   <commPort computRsc="controle3" computRscPort="ep_eth0" name="
      ↪ controle3_eth0"/>
190   <commPort computRsc="p6" computRscPort="ts_eth1" name="p6_eth1"/>
191  </commRsc>
192  <commRsc def="::ethernet_rsc::Ethernet_Link" name="controle4_to_p6">
193   <commPort computRsc="controle4" computRscPort="ep_eth0" name="
      ↪ controle4_eth0"/>
194   <commPort computRsc="p6" computRscPort="ts_eth2" name="p6_eth2"/>
195  </commRsc>
196  <commRsc def="::ethernet_rsc::Ethernet_Link" name="p6_to_p1">
197   <commPort computRsc="p6" computRscPort="ts_eth3" name="p6_eth3"/>
198   <commPort computRsc="p1" computRscPort="ts_eth3" name="p1_eth3"/>
199  </commRsc>
200 </environment>
```

Listing A.16 – Modèle de l'instanciation d'une topologie plus complexe.

A.3.2 Modèle des flux de données d'un réseau plus complexe

```
<?xml version="1.0" encoding="UTF-8"?>
<environment name="streams">
 <commRsc def="::tsn_rsc::Periodic_Stream" name="
   ↪ conn_passerelle_passerelle1_ordre_emtr_pe_to_mission1">
  <comment>stream1</comment>
  <config def="pcp" value="5"/>
  <config def="payload" value="{60, B}"/>
  <config def="offset" value="{2, us}"/>
  <config def="period" value="{180000, us}"/>
  <rscConfig def="talker" value="::model::topologie2::passerelle1_to_p5.
   ↪ passerelle1_eth0"/>
  <rscConfig def="listener" value="::model::topologie2::mission1_to_p5.
   ↪ mission1_eth0"/>
 </commRsc>
 <commRsc def="::tsn_rsc::Periodic_Stream" name="
   ↪ consigne_conn_controle1_consigne_moteur_emtr_pe_to_moteur1">
  <comment>stream2</comment>
  <config def="pcp" value="7"/>
  <config def="payload" value="{38, B}"/>
  <config def="offset" value="{11002, us}"/>
  <config def="period" value="{30000, us}"/>
  <rscConfig def="talker" value="::model::topologie2::controle1_to_p1.
   ↪ controle1_eth0"/>
  <rscConfig def="listener" value="::model::topologie2::moteur1_to_p3.
   ↪ moteur1_eth0"/>
 </commRsc>
 <commRsc def="::tsn_rsc::Periodic_Stream" name="
   ↪ etat_conn_moteur1_etat_emtr_pe_to_controle1">
  <comment>stream3</comment>
  <config def="pcp" value="7"/>
  <config def="payload" value="{44, B}"/>
  <config def="offset" value="{26004, us}"/>
  <config def="period" value="{30000, us}"/>
  <rscConfig def="talker" value="::model::topologie2::moteur1_to_p3.
   ↪ moteur1_eth0"/>
  <rscConfig def="listener" value="::model::topologie2::controle1_to_p1.
   ↪ controle1_eth0"/>
 </commRsc>
```

```
30  <commRsc def="::tsn_rsc::Periodic_Stream" name="
      ↪ consigne2_conn_controle2_consigne_moteur_emtr_pe_to_moteur2">
31   <comment>stream4</comment>
32   <config def="pcp" value="7"/>
33   <config def="payload" value="{38, B}"/>
34   <config def="offset" value="{11002, us}"/>
35   <config def="period" value="{30000, us}"/>
36   <rscConfig def="talker" value="::model::topologie2::controle2_to_p1.
      ↪ controle2_eth0"/>
37   <rscConfig def="listener" value="::model::topologie2::moteur2_to_p3.
      ↪ moteur2_eth0"/>
38  </commRsc>
39  <commRsc def="::tsn_rsc::Periodic_Stream" name="
      ↪ etat2_conn_moteur2_etat_emtr_pe_to_controle2">
40   <comment>stream5</comment>
41   <config def="pcp" value="7"/>
42   <config def="payload" value="{44, B}"/>
43   <config def="offset" value="{26004, us}"/>
44   <config def="period" value="{30000, us}"/>
45   <rscConfig def="talker" value="::model::topologie2::moteur2_to_p3.
      ↪ moteur2_eth0"/>
46   <rscConfig def="listener" value="::model::topologie2::controle2_to_p1.
      ↪ controle2_eth0"/>
47  </commRsc>
48  <commRsc def="::tsn_rsc::Periodic_Stream" name="
      ↪ consigne3_conn_controle3_consigne_moteur_emtr_pe_to_moteur3">
49   <comment>stream6</comment>
50   <config def="pcp" value="7"/>
51   <config def="payload" value="{38, B}"/>
52   <config def="offset" value="{11002, us}"/>
53   <config def="period" value="{30000, us}"/>
54   <rscConfig def="talker" value="::model::topologie2::controle3_to_p6.
      ↪ controle3_eth0"/>
55   <rscConfig def="listener" value="::model::topologie2::moteur3_to_p4.
      ↪ moteur3_eth0"/>
56  </commRsc>
```

```
57  <commRsc def="::tsn_rsc::Periodic_Stream" name="
      ↪ etat3_conn_moteur3_etat_emtr_pe_to_controle3">
58   <comment>stream7</comment>
59   <config def="pcp" value="7"/>
60   <config def="payload" value="{44, B}"/>
61   <config def="offset" value="{26004, us}"/>
62   <config def="period" value="{30000, us}"/>
63   <rscConfig def="talker" value="::model::topologie2::moteur3_to_p4.
      ↪ moteur3_eth0"/>
64   <rscConfig def="listener" value="::model::topologie2::controle3_to_p6.
      ↪ controle3_eth0"/>
65  </commRsc>
66  <commRsc def="::tsn_rsc::Periodic_Stream" name="
      ↪ consigne4_conn_controle4_consigne_moteur_emtr_pe_to_moteur4">
67   <comment>stream8</comment>
68   <config def="pcp" value="7"/>
69   <config def="payload" value="{38, B}"/>
70   <config def="offset" value="{11002, us}"/>
71   <config def="period" value="{30000, us}"/>
72   <rscConfig def="talker" value="::model::topologie2::controle4_to_p6.
      ↪ controle4_eth0"/>
73   <rscConfig def="listener" value="::model::topologie2::moteur4_to_p4.
      ↪ moteur4_eth0"/>
74  </commRsc>
75  <commRsc def="::tsn_rsc::Periodic_Stream" name="
      ↪ etat4_conn_moteur4_etat_emtr_pe_to_controle4">
76   <comment>stream9</comment>
77   <config def="pcp" value="7"/>
78   <config def="payload" value="{44, B}"/>
79   <config def="offset" value="{26004, us}"/>
80   <config def="period" value="{30000, us}"/>
81   <rscConfig def="talker" value="::model::topologie2::moteur4_to_p4.
      ↪ moteur4_eth0"/>
82   <rscConfig def="listener" value="::model::topologie2::controle4_to_p6.
      ↪ controle4_eth0"/>
83  </commRsc>
```

```
84  <commRsc def="::tsn_rsc::Periodic_Stream" name="
      ↪ boussole_conn_boussole1_orientation_emtr_pe_to_mission1">
85   <comment>stream10</comment>
86   <config def="pcp" value="4"/>
87   <config def="payload" value="{12, B}"/>
88   <config def="offset" value="{2002, us}"/>
89   <config def="period" value="{30000, us}"/>
90   <rscConfig def="talker" value="::model::topologie2::boussole_to_p2.
      ↪ boussole_eth0"/>
91   <rscConfig def="listener" value="::model::topologie2::mission1_to_p5.
      ↪ mission1_eth0"/>
92  </commRsc>
93  <commRsc def="::tsn_rsc::Periodic_Stream" name="
      ↪ position_conn_position1_position_emtr_pe_to_mission1">
94   <comment>stream11</comment>
95   <config def="pcp" value="4"/>
96   <config def="payload" value="{60, B}"/>
97   <config def="offset" value="{2002, us}"/>
98   <config def="period" value="{30000, us}"/>
99   <rscConfig def="talker" value="::model::topologie2::position_to_p2.
      ↪ position_eth0"/>
100  <rscConfig def="listener" value="::model::topologie2::mission1_to_p5.
      ↪ mission1_eth0"/>
101 </commRsc>
102 <commRsc def="::tsn_rsc::Periodic_Stream" name="
      ↪ ordre_mission_mission1_ordre_emtr_pe_to_controle2">
103  <comment>stream12</comment>
104  <config def="pcp" value="6"/>
105  <config def="payload" value="{60, B}"/>
106  <config def="offset" value="{10005, us}"/>
107  <config def="period" value="{30000, us}"/>
108  <rscConfig def="talker" value="::model::topologie2::mission1_to_p5.
      ↪ mission1_eth0"/>
109  <rscConfig def="listener" value="::model::topologie2::controle2_to_p1.
      ↪ controle2_eth0"/>
110 </commRsc>
```

```
111  <commRsc def="::tsn_rsc::Periodic_Stream" name="
       ↪ ordre_mission_mission1_ordre_emtr_pe_to_controle3">
112   <comment>stream13</comment>
113   <config def="pcp" value="6"/>
114   <config def="payload" value="{60, B}"/>
115   <config def="offset" value="{10005, us}"/>
116   <config def="period" value="{30000, us}"/>
117   <rscConfig def="talker" value="::model::topologie2::mission1_to_p5.
       ↪ mission1_eth0"/>
118   <rscConfig def="listener" value="::model::topologie2::controle3_to_p6.
       ↪ controle3_eth0"/>
119  </commRsc>
120  <commRsc def="::tsn_rsc::Periodic_Stream" name="
       ↪ ordre_mission_mission1_ordre_emtr_pe_to_controle4">
121   <comment>stream14</comment>
122   <config def="pcp" value="6"/>
123   <config def="payload" value="{60, B}"/>
124   <config def="offset" value="{10005, us}"/>
125   <config def="period" value="{30000, us}"/>
126   <rscConfig def="talker" value="::model::topologie2::mission1_to_p5.
       ↪ mission1_eth0"/>
127   <rscConfig def="listener" value="::model::topologie2::controle4_to_p6.
       ↪ controle4_eth0"/>
128  </commRsc>
129  <commRsc def="::tsn_rsc::Periodic_Stream" name="
       ↪ ordre_mission_mission1_ordre_emtr_pe_to_controle1">
130   <comment>stream15</comment>
131   <config def="pcp" value="6"/>
132   <config def="payload" value="{60, B}"/>
133   <config def="offset" value="{10005, us}"/>
134   <config def="period" value="{30000, us}"/>
135   <rscConfig def="talker" value="::model::topologie2::mission1_to_p5.
       ↪ mission1_eth0"/>
136   <rscConfig def="listener" value="::model::topologie2::controle1_to_p1.
       ↪ controle1_eth0"/>
137  </commRsc>
138 </environment>
```

Listing A.17 – Modèle de l'instanciation des flux de données d'une topologie plus complexe.

A.3.3 Modèle des exigences système d'un réseau plus complexe

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<environment name="app3_sr">
 <commRsc def="::tsn_rsc::Stream_System_Requirements" name="stream1">
  <rscConfig def="Stream" value="::streams::
   ↪ boussole_conn_boussole1_orientation_emtr_pe_to_mission1"/>
  <rscConfig def="overridden_talker" value="::model::topologie2::
   ↪ boussole_to_p2.boussole_eth0"/>
  <rscConfig def="overridden_listener" value="::model::topologie2::
   ↪ mission1_to_p5.mission1_eth0"/>
  <structConfig def="path">
   <rscConfig def="path_member" value="::model::topologie2::
   ↪ boussole_to_p2.p2_eth1"/>
   <rscConfig def="path_member" value="::model::topologie2::p2_to_p3.
   ↪ p3_eth0"/>
   <rscConfig def="path_member" value="::model::topologie2::p3_to_p4.
   ↪ p4_eth0"/>
   <rscConfig def="path_member" value="::model::topologie2::p4_to_p5.
   ↪ p5_eth0"/>
  </structConfig>
  <config def="vlan_id" value="1"/>
  <config def="deadline" value="{500,ms}"/>
  <config def="is_critical" value="false"/>
 </commRsc>
 <commRsc def="::tsn_rsc::Stream_System_Requirements" name="stream2">
  <rscConfig def="Stream" value="::streams::
   ↪ conn_passerelle_passerelle1_ordre_emtr_pe_to_mission1"/>
  <rscConfig def="overridden_talker" value="::model::topologie2::
   ↪ passerelle1_to_p5.passerelle1_eth0"/>
  <rscConfig def="overridden_listener" value="::model::topologie2::
   ↪ mission1_to_p5.mission1_eth0"/>
  <structConfig def="path">
   <rscConfig def="path_member" value="::model::topologie2::
   ↪ passerelle1_to_p5.p5_eth2"/>
  </structConfig>
  <config def="vlan_id" value="1"/>
  <config def="deadline" value="{250,ms}"/>
  <config def="is_critical" value="false"/>
 </commRsc>
```

```
28  <commRsc def="::tsn_rsc::Stream_System_Requirements" name="stream3">
29   <rscConfig def="Stream" value="::streams::
      ↪ consigne_conn_controle1_consigne_moteur_emtr_pe_to_moteur1"/>
30   <rscConfig def="overridden_talker" value="::model::topologie2::
      ↪ controle1_to_p1.controle1_eth0"/>
31   <rscConfig def="overridden_listener" value="::model::topologie2::
      ↪ moteur1_to_p3.moteur1_eth0"/>
32   <structConfig def="path">
33    <rscConfig def="path_member" value="::model::topologie2::
      ↪ controle1_to_p1.p1_eth0"/>
34    <rscConfig def="path_member" value="::model::topologie2::p1_to_p2.
      ↪ p2_eth0"/>
35    <rscConfig def="path_member" value="::model::topologie2::p2_to_p3.
      ↪ p3_eth0"/>
36   </structConfig>
37   <config def="vlan_id" value="1"/>
38   <config def="deadline" value="{10,ms}"/>
39   <config def="maximum_jitter" value="{1,ms}"/>
40   <config def="is_critical" value="true"/>
41  </commRsc>
42  <commRsc def="::tsn_rsc::Stream_System_Requirements" name="stream4">
43   <rscConfig def="Stream" value="::streams::
      ↪ consigne2_conn_controle2_consigne_moteur_emtr_pe_to_moteur2"/>
44   <rscConfig def="overridden_talker" value="::model::topologie2::
      ↪ controle2_to_p1.controle2_eth0"/>
45   <rscConfig def="overridden_listener" value="::model::topologie2::
      ↪ moteur2_to_p3.moteur2_eth0"/>
46   <structConfig def="path">
47    <rscConfig def="path_member" value="::model::topologie2::
      ↪ controle2_to_p1.p1_eth1"/>
48    <rscConfig def="path_member" value="::model::topologie2::p1_to_p2.
      ↪ p2_eth0"/>
49    <rscConfig def="path_member" value="::model::topologie2::p2_to_p3.
      ↪ p3_eth0"/>
50   </structConfig>
51   <config def="vlan_id" value="1"/>
52   <config def="deadline" value="{10,ms}"/>
53   <config def="maximum_jitter" value="{1,ms}"/>
54   <config def="is_critical" value="true"/>
55  </commRsc>
```

```
56  <commRsc def="::tsn_rsc::Stream_System_Requirements" name="stream5">
57   <rscConfig def="Stream" value="::streams::
      ↪ consigne3_conn_controle3_consigne_moteur_emtr_pe_to_moteur3"/>
58   <rscConfig def="overridden_talker" value="::model::topologie2::
      ↪ controle3_to_p6.controle3_eth0"/>
59   <rscConfig def="overridden_listener" value="::model::topologie2::
      ↪ moteur3_to_p4.moteur3_eth0"/>
60   <structConfig def="path">
61    <rscConfig def="path_member" value="::model::topologie2::
      ↪ controle3_to_p6.p6_eth1"/>
62    <rscConfig def="path_member" value="::model::topologie2::p5_to_p6.
      ↪ p5_eth3"/>
63    <rscConfig def="path_member" value="::model::topologie2::p4_to_p5.
      ↪ p4_eth3"/>
64   </structConfig>
65   <config def="vlan_id" value="1"/>
66   <config def="deadline" value="{10,ms}"/>
67   <config def="maximum_jitter" value="{1,ms}"/>
68   <config def="is_critical" value="true"/>
69  </commRsc>
70  <commRsc def="::tsn_rsc::Stream_System_Requirements" name="stream6">
71   <rscConfig def="Stream" value="::streams::
      ↪ consigne4_conn_controle4_consigne_moteur_emtr_pe_to_moteur4"/>
72   <rscConfig def="overridden_talker" value="::model::topologie2::
      ↪ controle4_to_p6.controle4_eth0"/>
73   <rscConfig def="overridden_listener" value="::model::topologie2::
      ↪ moteur4_to_p4.moteur4_eth0"/>
74   <structConfig def="path">
75    <rscConfig def="path_member" value="::model::topologie2::
      ↪ controle4_to_p6.p6_eth2"/>
76    <rscConfig def="path_member" value="::model::topologie2::p5_to_p6.
      ↪ p5_eth3"/>
77    <rscConfig def="path_member" value="::model::topologie2::p4_to_p5.
      ↪ p4_eth3"/>
78   </structConfig>
79   <config def="vlan_id" value="1"/>
80   <config def="deadline" value="{10,ms}"/>
81   <config def="maximum_jitter" value="{1,ms}"/>
82   <config def="is_critical" value="true"/>
83  </commRsc>
```

```
84  <commRsc def="::tsn_rsc::Stream_System_Requirements" name="stream7">
85   <rscConfig def="Stream" value="::streams::
      ↪ etat_conn_moteur1_etat_emtr_pe_to_controle1"/>
86   <rscConfig def="overridden_talker" value="::model::topologie2::
      ↪ moteur1_to_p3.moteur1_eth0"/>
87   <rscConfig def="overridden_listener" value="::model::topologie2::
      ↪ controle1_to_p1.controle1_eth0"/>
88   <structConfig def="path">
89    <rscConfig def="path_member" value="::model::topologie2::
      ↪ moteur1_to_p3.p3_eth1"/>
90    <rscConfig def="path_member" value="::model::topologie2::p2_to_p3.
      ↪ p2_eth3"/>
91    <rscConfig def="path_member" value="::model::topologie2::p1_to_p2.
      ↪ p1_eth2"/>
92   </structConfig>
93   <config def="vlan_id" value="1"/>
94   <config def="deadline" value="{10,ms}"/>
95   <config def="maximum_jitter" value="{1,ms}"/>
96   <config def="is_critical" value="true"/>
97  </commRsc>
98  <commRsc def="::tsn_rsc::Stream_System_Requirements" name="stream8">
99   <rscConfig def="Stream" value="::streams::
      ↪ etat2_conn_moteur2_etat_emtr_pe_to_controle2"/>
100  <rscConfig def="overridden_talker" value="::model::topologie2::
      ↪ moteur2_to_p3.moteur2_eth0"/>
101  <rscConfig def="overridden_listener" value="::model::topologie2::
      ↪ controle2_to_p1.controle2_eth0"/>
102  <structConfig def="path">
103   <rscConfig def="path_member" value="::model::topologie2::
      ↪ moteur2_to_p3.p3_eth2"/>
104   <rscConfig def="path_member" value="::model::topologie2::p2_to_p3.
      ↪ p2_eth3"/>
105   <rscConfig def="path_member" value="::model::topologie2::p1_to_p2.
      ↪ p1_eth2"/>
106  </structConfig>
107  <config def="vlan_id" value="1"/>
108  <config def="deadline" value="{10,ms}"/>
109  <config def="maximum_jitter" value="{1,ms}"/>
110  <config def="is_critical" value="true"/>
111 </commRsc>
```

```
112  <commRsc def="::tsn_rsc::Stream_System_Requirements" name="stream9">
113   <rscConfig def="Stream" value="::streams::
       ↪ etat3_conn_moteur3_etat_emtr_pe_to_controle3"/>
114   <rscConfig def="overridden_talker" value="::model::topologie2::
       ↪ moteur3_to_p4.moteur3_eth0"/>
115   <rscConfig def="overridden_listener" value="::model::topologie2::
       ↪ controle3_to_p6.controle3_eth0"/>
116   <structConfig def="path">
117    <rscConfig def="path_member" value="::model::topologie2::
       ↪ moteur3_to_p4.p4_eth1"/>
118    <rscConfig def="path_member" value="::model::topologie2::p4_to_p5.
       ↪ p5_eth0"/>
119    <rscConfig def="path_member" value="::model::topologie2::p5_to_p6.
       ↪ p6_eth0"/>
120   </structConfig>
121   <config def="vlan_id" value="1"/>
122   <config def="deadline" value="{10,ms}"/>
123   <config def="maximum_jitter" value="{1,ms}"/>
124   <config def="is_critical" value="true"/>
125  </commRsc>
126  <commRsc def="::tsn_rsc::Stream_System_Requirements" name="stream10">
127   <rscConfig def="Stream" value="::streams::
       ↪ etat4_conn_moteur4_etat_emtr_pe_to_controle4"/>
128   <rscConfig def="overridden_talker" value="::model::topologie2::
       ↪ moteur4_to_p4.moteur4_eth0"/>
129   <rscConfig def="overridden_listener" value="::model::topologie2::
       ↪ controle4_to_p6.controle4_eth0"/>
130   <structConfig def="path">
131    <rscConfig def="path_member" value="::model::topologie2::
       ↪ moteur4_to_p4.p4_eth2"/>
132    <rscConfig def="path_member" value="::model::topologie2::p4_to_p5.
       ↪ p5_eth0"/>
133    <rscConfig def="path_member" value="::model::topologie2::p5_to_p6.
       ↪ p6_eth0"/>
134   </structConfig>
135   <config def="vlan_id" value="1"/>
136   <config def="deadline" value="{10,ms}"/>
137   <config def="maximum_jitter" value="{1,ms}"/>
138   <config def="is_critical" value="true"/>
139  </commRsc>
```

```
140  <commRsc def="::tsn_rsc::Stream_System_Requirements" name="stream11">
141   <rscConfig def="Stream" value="::streams::
       ↪ ordre_mission_mission1_ordre_emtr_pe_to_controle1"/>
142   <rscConfig def="overridden_talker" value="::model::topologie2::
       ↪ mission1_to_p5.mission1_eth0"/>
143   <rscConfig def="overridden_listener" value="::model::topologie2::
       ↪ controle1_to_p1.controle1_eth0"/>
144   <structConfig def="path">
145    <rscConfig def="path_member" value="::model::topologie2::
       ↪ mission1_to_p5.p5_eth1"/>
146    <rscConfig def="path_member" value="::model::topologie2::p5_to_p6.
       ↪ p6_eth0"/>
147    <rscConfig def="path_member" value="::model::topologie2::p6_to_p1.
       ↪ p1_eth3"/>
148   </structConfig>
149   <config def="vlan_id" value="1"/>
150   <config def="deadline" value="{100,ms}"/>
151   <config def="is_critical" value="false"/>
152  </commRsc>
153  <commRsc def="::tsn_rsc::Stream_System_Requirements" name="stream12">
154   <rscConfig def="Stream" value="::streams::
       ↪ ordre_mission_mission1_ordre_emtr_pe_to_controle2"/>
155   <rscConfig def="overridden_talker" value="::model::topologie2::
       ↪ mission1_to_p5.mission1_eth0"/>
156   <rscConfig def="overridden_listener" value="::model::topologie2::
       ↪ controle2_to_p1.controle2_eth0"/>
157   <structConfig def="path">
158    <rscConfig def="path_member" value="::model::topologie2::
       ↪ mission1_to_p5.p5_eth1"/>
159    <rscConfig def="path_member" value="::model::topologie2::p5_to_p6.
       ↪ p6_eth0"/>
160    <rscConfig def="path_member" value="::model::topologie2::p6_to_p1.
       ↪ p1_eth3"/>
161   </structConfig>
162   <config def="vlan_id" value="1"/>
163   <config def="deadline" value="{100,ms}"/>
164   <config def="is_critical" value="false"/>
165  </commRsc>
```

```
166  <commRsc def="::tsn_rsc::Stream_System_Requirements" name="stream13">
167   <rscConfig def="Stream" value="::streams::
       ↪ ordre_mission_mission1_ordre_emtr_pe_to_controle3"/>
168   <rscConfig def="overridden_talker" value="::model::topologie2::
       ↪ mission1_to_p5.mission1_eth0"/>
169   <rscConfig def="overridden_listener" value="::model::topologie2::
       ↪ controle3_to_p6.controle3_eth0"/>
170   <structConfig def="path">
171    <rscConfig def="path_member" value="::model::topologie2::
       ↪ mission1_to_p5.p5_eth1"/>
172    <rscConfig def="path_member" value="::model::topologie2::p5_to_p6.
       ↪ p6_eth0"/>
173   </structConfig>
174   <config def="vlan_id" value="1"/>
175   <config def="deadline" value="{100,ms}"/>
176   <config def="is_critical" value="false"/>
177  </commRsc>
178  <commRsc def="::tsn_rsc::Stream_System_Requirements" name="stream14">
179   <rscConfig def="Stream" value="::streams::
       ↪ ordre_mission_mission1_ordre_emtr_pe_to_controle4"/>
180   <rscConfig def="overridden_talker" value="::model::topologie2::
       ↪ mission1_to_p5.mission1_eth0"/>
181   <rscConfig def="overridden_listener" value="::model::topologie2::
       ↪ controle4_to_p6.controle4_eth0"/>
182   <structConfig def="path">
183    <rscConfig def="path_member" value="::model::topologie2::
       ↪ mission1_to_p5.p5_eth1"/>
184    <rscConfig def="path_member" value="::model::topologie2::p5_to_p6.
       ↪ p6_eth0"/>
185   </structConfig>
186   <config def="vlan_id" value="1"/>
187   <config def="deadline" value="{100,ms}"/>
188   <config def="is_critical" value="false"/>
189  </commRsc>
```

```
190   <commRsc def="::tsn_rsc::Stream_System_Requirements" name="stream15">
191    <rscConfig def="Stream" value="::streams::
       ↪ position_conn_position1_position_emtr_pe_to_mission1"/>
192    <rscConfig def="overridden_talker" value="::model::topologie2::
       ↪ position_to_p2.position_eth0"/>
193    <rscConfig def="overridden_listener" value="::model::topologie2::
       ↪ mission1_to_p5.mission1_eth0"/>
194    <structConfig def="path">
195     <rscConfig def="path_member" value="::model::topologie2::
       ↪ position_to_p2.p2_eth2"/>
196     <rscConfig def="path_member" value="::model::topologie2::p2_to_p3.
       ↪ p3_eth0"/>
197     <rscConfig def="path_member" value="::model::topologie2::p3_to_p4.
       ↪ p4_eth0"/>
198     <rscConfig def="path_member" value="::model::topologie2::p4_to_p5.
       ↪ p5_eth0"/>
199    </structConfig>
200    <config def="vlan_id" value="1"/>
201    <config def="deadline" value="{500,ms}"/>
202    <config def="is_critical" value="false"/>
203   </commRsc>
204 </environment>
```

Listing A.18 – Modèle de l'instanciation des exigences du systèmes d'une topologie plus complexe.

Annexe B

Plateforme Pharos

Comme nous l'avons expliqué à la section 3.2.3, une spécification d'architecture logicielle en UCM repose sur la définition d'une *plateforme*. Une plateforme est un ensemble de politiques techniques et de connecteur qui matérialisent le code technique prenant en charge l'exécution des composants (par les politiques techniques) et les communications (par les connecteurs) au sein de l'application

Le développement d'une application répartie dont les communications sont prises en charge par un réseau TSN nécessite de respecter certaines contraintes, typiquement : des tailles de données bornées et des instants d'émission de données déterministes. L'architecture d'une telle application doit ainsi pouvoir être réalisée de telle sorte qu'elle garantisse le respect de ces contraintes.

Au cours de nos travaux, une plateforme UCM a été spécifiée et réalisée au sein de l'équipe pour les besoins d'expérimentations. Cette plateforme a été appelée « Pharos ». Nous avons participé à sa conception pour s'assurer qu'il soit possible d'extraire les informations nécessaires à la modélisation du réseau et le calcul des flux (cf. chapitres 5 et 6). Dans cette annexe nous décrivons les éléments de cette plateforme.

B.1 Contraintes liées au type de systèmes cible

Dans nos travaux, nous considérons des applications réparties dont les différents nœuds communiquent par un réseau Ethernet. Les contraintes liées aux réseaux temps réels conduisent à l'utilisation de sockets UDP plutôt que TCP. En effet, le protocole TCP peut engendrer des retransmissions ainsi que des acquittements, ce qui rend difficile la caractérisation précise des flux réseau ; à l'inverse, le protocole UDP n'engendre pas de flux supplémentaire.

Par ailleurs, afin d'illustrer l'impact des modifications d'architecture logicielle, nous choisissons de considérer la possibilité d'utiliser des communications signées (selon la signature RSA4096) ou non. Cela permet de montrer l'influence de la configuration des couches de communication sur les flux de données.

Enfin, nous choisissons de sérialiser les données au format CBOR [CBO13]. CBOR est bien adapté aux communications temps-réel car il est facile de calculer la taille maximale de la sérialisation pour une structure de donnée dont la taille est connue. De plus, sa mise en œuvre par la bibliothèque `libcbor` est simple à utiliser.

B.2 La plateforme Pharos

La plateforme Pharos est constituée d'un ensemble de définitions de ressources, de connecteurs et de politiques techniques. Les ressources sont utilisées pour modéliser la topologie réseau et les flux de données dans le chapitre 5 tandis que les connecteurs et les politiques techniques permettent de calculer les caractéristiques des flux dans le chapitre 6.

B.2.1 Ressources

La plateforme Pharos définit un ensemble de ressources permettant de décrire la topologie réseau et de saisir les différentes informations nécessaires à la caractérisation de la configuration réseau ainsi que des flux de données. Ces ressources sont organisées en plusieurs modules.

Un module appelé *ethernet_rsc* définit les notions de lien Ethernet, de terminal et de commutateur (non TSN). Les principales déclarations en sont représentées au listing B.19.

```
</resourceModule name="ethernet_rsc">
 [...]
 <commRscDef name="Ethernet_Link">
  <commRscPort def="Ethernet_interface" max="2" min="2" name="
   ↪ ethernet_interfaces_el"/>
  <configParam name="propagation_delay" type="::tsn_rsc::types::time_t"/
   ↪ >
 </commRscDef>
 <commPortDef name="Ethernet_interface">
  <configParam name="bandwidth" type="types::bandwidth_t"/>
 </commPortDef>
 <commPortDef name="Eth_IP_interface">
  <extends ref="Ethernet_interface"/>
  <configParam name="IP_address" type="types::IP_t"/>
  <configParam name="MAC_address" type="types::MAC_address_t"/>
 </commPortDef>
 <computRscDef name="End_Point">
  <computRscPort def="Eth_IP_interface" max="1" min="0" name="ep_eth0"/>
  [...]
 </computRscDef>
 [...]
</resourceModule>
```

Listing B.19 – Définition des ressources pour les réseaux Ethernet.

Ces définitions permettent de construire une topologie Ethernet « classique », c'est-à-dire de décrire un réseau Ethernet commuté sans les caractéristiques propres à TSN. La spécification des paramètres liés à TSN est portée par un ensemble de définitions de ressources, qui viennent compléter les définitions Ethernet de base.

Un autre module appelé *tsn_rsc* contient ainsi les définitions propres aux réseaux TSN. Cela comprend notamment les commutateurs TSN à 3, 4, 6 et 8 ports [23]. Ces définitions figurent au listing B.20. Les définitions des commutateurs TSN identifient séparément chaque port Ethernet.

23. Le choix de définir plusieurs versions de commutateurs plutôt qu'une seule ayant un nombre de ports variables a été fait pour des raisons pratiques de modélisation. En effet, chaque port doit pouvoir être identifié individuellement pour faciliter la modélisation de la topologie. Il est très facile de définir de nouvelles versions de commutateurs avec un nombre de ports différents.

En effet, chaque port Ethernet peut avoir une configuration du *Time Aware Shaper* et du *Credit Based Shaper* différente.

```
<resourceModule name="tsn_rsc">
 [...]
 <computRscDef name="TSN_Switch">
  <computRscPort def="::ethernet_rsc::Ethernet_interface" max="1" min="0
   ↪ " name="ts_eth0"/>
  <computRscPort def="::ethernet_rsc::Ethernet_interface" max="1" min="0
   ↪ " name="ts_eth1"/>
  <computRscPort def="::ethernet_rsc::Ethernet_interface" max="1" min="0
   ↪ " name="ts_eth2"/>
  <configParam max="-1" min="0" name="supported_functionalities" type="
   ↪ types::supported_functionalities"/>
  <configParam name="processing_delay" type="types::time_t"/>
 </computRscDef>
 <computRscDef name="TSN_Switch_4_Ports">
  <extends ref="TSN_Switch"/>
  <computRscPort def="::ethernet_rsc::Ethernet_interface" max="1" min="0
   ↪ " name="ts_eth3"/>
 </computRscDef>
 <computRscDef name="TSN_Switch_6_Ports">
  <extends ref="TSN_Switch_4_Ports"/>
  <computRscPort def="::ethernet_rsc::Ethernet_interface" max="1" min="0
   ↪ " name="ts_eth4"/>
  <computRscPort def="::ethernet_rsc::Ethernet_interface" max="1" min="0
   ↪ " name="ts_eth5"/>
 </computRscDef>
 <computRscDef name="TSN_Switch_8_Ports">
  <extends ref="TSN_Switch_6_Ports"/>
  <computRscPort def="::ethernet_rsc::Ethernet_interface" max="1" min="0
   ↪ " name="ts_eth6"/>
  <computRscPort def="::ethernet_rsc::Ethernet_interface" max="1" min="0
   ↪ " name="ts_eth7"/>
 </computRscDef>
</resourceModule>
```

Listing B.20 – Définition des ressources pour les commutateurs TSN.

Le module *tsn_rsc* contient aussi les définitions des flux de données, représentés au listing B.21. Trois types de flux sont définis : flux périodique (*periodic stream*), apériodique (*aperiodic stream*) et sporadique (*sporadic stream*). Tous les trois ont pour ancêtre le flux applicatif (*applicative stream*), qui rassemble les paramètres de configuration communs.

Un flux périodique se caractérise par une émission répétée et régulière de données. Un flux sporadique correspond à une émission répétée mais irrégulière de donnée, avec une période minimale d'interarrivés (c'est-à-dire que de nouvelles données ne peuvent pas être émises avant l'expiration de la période minimale). Un flux apériodique est une émission de données unique, qui ne se répète pas.

Comme décrit dans le chapitre 5, les caractéristiques communes à toutes les définitions de flux sont la taille des données transportées, le niveau de priorité (PCP), la phase d'émission (c'est-à-dire le décalage par rapport au démarrage du système), la date de fin d'émission du flux, l'émetteur et le récepteur. La date de fin d'émission permet d'indiquer que le flux n'émettra plus au delà d'un certain moment.

```
<resourceModule name="tsn_rsc">
 [...]
 <commRscDef name="Applicative_Stream">
  <configParam name="payload" type="types::payload_t"/>
  <configParam name="pcp" type="types::pcp_t"/>
  <configParam min="0" name="offset" type="types::time_t"/>
  <configParam min="0" name="end_time" type="types::time_t"/>
  <rscParam max="1" min="0" name="listener">
   <allowedRscDef ref="::ethernet_rsc::Eth_IP_interface"/>
  </rscParam>
  <rscParam max="1" min="0" name="talker">
   <allowedRscDef ref="::ethernet_rsc::Eth_IP_interface"/>
  </rscParam>
 </commRscDef>
 <commRscDef name="Periodic_Stream">
  <extends ref="Applicative_Stream"/>
  <configParam name="period" type="types::time_t"/>
 </commRscDef>
 <commRscDef name="Sporadic_Stream">
  <extends ref="Applicative_Stream"/>
  <configParam name="min_interval" type="types::time_t"/>
 </commRscDef>
 <commRscDef name="Aperiodic_Stream">
  <extends ref="Applicative_Stream"/>
 </commRscDef>
 [...]
</resourceModule>
```

Listing B.21 – Définition des ressources pour les flux.

Nous n'avons pas défini de flux en rafale (*burst stream*). En effet, bien que cette notion soit courante en réseau, notre travail s'intéresse spécifiquement aux flux produits par des applications de contrôle-commande. Ces applications ne produisent typiquement pas de flux en rafale.

Finalement, le module *tsn_rsc* contient également la définition des exigences systèmes (*system requirements*) que l'on associe aux flux. Une exigence système fait référence à un ou plusieurs flux applicatifs (c'est-à-dire un ou plusieurs flux périodiques, sporadiques ou apériodiques). Elle définit une date limite d'arrivée (l'échéance) pour ces flux, c'est-à-dire un délai maximum entre la date d'émission et la date de réception. Elle spécifie également une fluctuation maximale dans le délai de transmission des flux. Enfin, une exigence système permet d'indiquer un numéro de réseau virtuel (*VLAN*) et d'éventuellement spécifier l'émetteur et le récepteur, pour le cas où ceux-ci n'auraient pas été définis dans la déclaration des flux.

```
<resourceModule name="tsn_rsc">
 [...]
 <commRscDef name="Stream_System_Requirements">
  <rscParam max="-1" min="1" name="Stream">
   <allowedRscDef ref="Applicative_Stream"/>
  </rscParam>
  <configParam min="0" name="deadline" type="types::time_t"/>
  <configParam name="vlan_id" type="types::vlan_id_t"/>
  <configParam min="0" name="maximum_jitter" type="types::time_t"/>
 </commRscDef>
 [...]
</resourceModule>
```

Listing B.22 – Définition des ressources pour les exigences système.

Les applications réparties que nous considérons sont caractérisées par des communications en point à point, pour lesquelles il est nécessaire de connaître l'adresse IP et le port UDP du destinataire. Les nœuds de l'application logicielle doivent donc être déployés explicitement sur les terminaux de la topologie réseau afin de déterminer les adresses IP. Cela nécessite la définition de nœuds de composants pour lesquels on peut spécifier l'allocation sur les terminaux.

Le listing B.23 contient la définition `prs_process` correspondant à de tels nœuds. Un nœud de composant `prs_process` possède deux paramètres : un paramètre d'allocation sur un terminal (*End_Point*) qui porte les adresses IP (comme présenté dans le listing B.19) et un paramètre pour spécifier le port UDP d'écoute.

```
<platformModule name="pharos_lib">
 [...]
 <resourceModule name="rsc">
  <contractModule name="config">
   <integer kind="int32" name="udp_port_t"/>
   [...]
  </contractModule>
  <compNodeDef name="prs_process">
   <rscParam max="1" min="1" name="hw_allocation">
    <allowedRscDef ref="::ethernet_rsc::End_Point"/>
   </rscParam>
   <configParam name="udp_port" type="config::udp_port_t">
   </configParam>
  </compNodeDef>
  [...]
 </resourceModule>
 [...]
</platformModule>
```

Listing B.23 – Définitions des nœuds.

B.2.2 Politiques techniques

La plateforme Pharos définit cinq politiques techniques qui reprennent directement des politiques techniques définies dans la bibliothèque standard d'UCM [UCM21] : exécution passive non protégée des composants, politique de *log*, interface de programmation pour obtenir l'horloge système, politiques de déclenchement périodique et unique des composants.

Les deux politiques techniques de déclenchement affectent directement la sémantique d'exécution des composants et donc le calcul du modèle des flux de données effectué au chapitre 6. Leurs définitions sont reproduites au listing B.24. Les deux politiques techniques `prs_periodic` et `prs_single` étendent les deux définitions du standard UCM `prdc_self_exec_comp` et `bgnd_self_exec_comp` et en précisent la sémantique : chaque instance de politique technique est associée à un fil d'exécution (*thread*) POSIX qui assure le déclenchement – périodique ou unique – du code appelé par la politique.

```
<platformModule name="pharos_lib">
 [...]
 <policyModule name="trig">
  [...]
  <policyDef applicability="on_component_only" aspect="::ucm_core::
   ↪ comp_exec::comp_trig_asp" name="prs_periodic">
   <extends ref="::ucm_ext_exec::prdc_self_exec_comp"/>
  </policyDef>
  <policyDef applicability="on_component_only" aspect="::ucm_core::
   ↪ comp_exec::comp_trig_asp" name="prs_single">
   <extends ref="::ucm_ext_exec::bgnd_self_exec_comp"/>
  </policyDef>
 </policyModule>
 [...]
</platformModule>
```

Listing B.24 – Définitions des politiques techniques d'exécution de la plateforme Pharos.

L'utilisation d'un fil d'exécution dédié pour chacune des instances de politiques techniques de déclenchement implique la possibilité d'exécutions concurrentes au sein de l'application, à condition que les systèmes d'exploitation et les processeurs sous-jacents soient capables d'assurer ce parallélisme. D'un point de vue sémantique, et dans le cadre de nos travaux, nous considérons que deux politiques d'exécution périodique spécifiant des exécutions ayant lieu en même temps vont effectivement engendrer des exécutions simultanées.

B.2.3 Connecteurs

La plateforme Pharos définit plusieurs connecteurs qui reprennent les interfaces de la bibliothèque standard UCM en en raffinent la sémantique. Dans le cadre des travaux de cette thèse, les quatre les plus pertinents sont : l'envoi de message par *socket unicast* avec sérialisation CBOR, l'envoi de message par *socket unicast* avec sérialisation CBOR et chiffrement RSAS4096, la donnée partagée par *socket unicast* avec sérialisation CBOR et la donnée partagée par *socket unicast* avec sérialisation CBOR et chiffrement RSA4096.

La bibliothèque Pharos fournit trois connecteurs pour les communications locales directes entre composants, sans passer par le réseau. Ces connecteurs étendent directement les définitions de la bibliothèque standard UCM en en précisant la sémantique : les communications se font directement en transmettant les appels de méthode.

Les trois connecteurs permettent de réaliser respectivement des envoi de message, du partage de donnée et de l'appel de service. L'envoi de message consiste à envoyer une donnée d'un émetteur vers des récepteurs ; l'arrivée du message déclenche sa consommation par les récepteurs indépendamment les uns des autres. La donnée partagée consiste à écrire une donnée et à notifier des lecteurs ; les lecteurs peuvent à tout moment lire la donnée, sans la consommer : ils peuvent ainsi lire plusieurs fois la même donnée, ou ne pas la lire alors qu'ils ont été notifiés. L'appel de

service correspond à un appel de méthode.

```
<platformModule name="pharos_lib">
 [...]
 <interactionModule name="local_comm">
  <connectorDef name="prs_local_msg_cnt" pattern="::ucm_core::messages::
   ↪ msg_intr_pat">
   <comment>intra-process communication</comment>
   <extends ref="::ucm_core::messages::simple_msg_cnt"/>
  </connectorDef>
  <connectorDef name="prs_local_sd_cnt" pattern="::ucm_ext_interac::
   ↪ shared_data::sd_intr_pat">
   <comment>intra-process communication</comment>
   <extends ref="::ucm_ext_interac::shared_data::sd_cnt"/>
  </connectorDef>
  <connectorDef name="prs_local_svc_cnt" pattern="::ucm_core::services::
   ↪ svc_intr_pat">
   <extends ref="::ucm_core::services::simple_svc_cnt"/>
  </connectorDef>
 </interactionModule>
 [...]
</platformModule>
```

Listing B.25 – Définitions des connecteurs pour les communications locales (sans passer par le réseau).

La définition du connecteur d'envoi de message par *socket* `prs_socket_msg_cnt` est représentée au listing B.26. Elle en reprend la sémantique, à savoir qu'une donnée est envoyée par un émetteur à des récepteurs, et que la réception du message déclenche la consommation de la donnée par chaque récepteur, indépendamment.

La définition du connecteur de message par *socket* étend le connecteur de message local et définit les paramètres nécessaires à sa configuration. Le connecteur permet ainsi d'indiquer sur quel port Ethernet les données doivent arriver, afin de calculer l'adresse IP de destination (paramètre `network_interface`).

Également, il permet de spécifier le niveau de priorité (PCP) qui sera associé aux communications. Le connecteur permet de spécifier une priorité générale (paramètre `default_priority`), ansi que des PCP spécifiques pour un couple (émetteur, récepteur) donné ; pour cela, il est nécessaire de renseigner les valeurs des paramètres `comm_set_id` des ports d'émission et de réception, et d'indiquer la valeur de PCP pour les couples (émetteur, récepteur) en utilisant les paramètres `src_set_id`, `dst_set_id` et `priority`.

Les communications réalisées par ce connecteur sont systématiquement réalisées au moyen de *socket* UDP unicast – même si l'émetteur et le récepteur sont sur le même nœud – avec une sérialisation des données en CBOR.

De façon générale, un message UDP est envoyé pour chaque récepteur. Néanmoins, si plusieurs récepteurs sont déployés sur le même nœud (`prs_process`) et que les valeurs de priorité sont les mêmes pour tous, alors un seul message UDP est envoyé sur le réseau à destination du terminal.

Le connecteur de donnée partagée par *socket* est défini selon le même principe.

```
<platformModule name="pharos_lib">
 [...]
 <interactionModule name="socket_comm">
  <contractModule name="types">
   <integer kind="int32" name="set_id_t"/>
   <integer kind="int32" name="connection_id_t"/>
  </contractModule>
  <connectorDef name="prs_socket_msg_cnt" pattern="::ucm_core::messages
   ↪ ::msg_intr_pat">
   <extends ref="::pharos_lib::local_comm::prs_local_msg_cnt"/>
   <configParam defaultValue="0" min="0" name="default_priority" type="
   ↪ ::ucm_ext_exec::contracts::priority_t"/>
   <structParam max="-1" min="0" name="specific_priority">
    <configParam name="src_set_id" type="types::set_id_t"/>
    <configParam name="dst_set_id" type="types::set_id_t"/>
    <configParam name="priority" type="::ucm_ext_exec::contracts::
   ↪ priority_t"/>
   </structParam>
   <configParam min="0" name="connection_id" type="types::
   ↪ connection_id_t"/>
   <portConf port="receiver">
    <rscParam max="1" min="0" name="network_interface">
     <allowedRscDef ref="::ethernet_rsc::Eth_IP_interface"/>
    </rscParam>
    <configParam min="0" name="comm_set_id" type="types::set_id_t"/>
   </portConf>
   <portConf port="emitter">
    <configParam min="0" name="comm_set_id" type="types::set_id_t"/>
   </portConf>
  </connectorDef>
 [...]
</platformModule>
```

Listing B.26 – Définitions des connecteurs pour l'envoi de message par socket.

La définition du connecteur de message avec chiffrement RSA4096 étend celle du connecteur de message *socket*, dont elle reprend tous les paramètres de configuration. Ce connecteur calcule selon le protocole RSA4096 une signature de la donnée sérialisée en CBOR et envoie le tout sur le réseau. Cette signature est effectuée à l'aide d'une clé de chiffrement qu'il faut fournir au connecteur lors de son initialisation. À la réception, le connecteur vérifie la signature et ne transmet la donnée que si cette vérification est correcte.

Chaque émetteur peut avoir une clé de chiffrement différente. Le récepteur doit avoir une clé de déchiffrement pour chacune des clés de chiffrement. Pour que le récepteur puisse sélectionner la bonne clé de déchiffrement, il faut pouvoir différencier les émetteurs. Cela se fait par le paramètre `eid`, qui est un nombre entier associé à chaque émetteur. Lors de l'initialisation de chaque récepteur, il est nécessaire de fournir (programatiquement) la clé de chiffrement ; lors de l'initialisation du récepteur, il est nécessaire de fournir des couples (identificateur, clé de déchiffrement).

```
<platformModule name="pharos_lib">
 [...]
 <interactionModule name="auth_comm">
  <connectorDef name="prs_rsa4096_msg_cnt" pattern="::ucm_core::messages
   ↪ ::msg_intr_pat">
   <comment>Message connector with RSA4096 data signature</comment>
   <extends ref="::pharos_lib::socket_comm::prs_socket_msg_cnt"/>
   [...]
   <portConf port="emitter">
    <configParam name="eid" type="types::emitter_id"/>
   </portConf>
   [...]
  </connectorDef>
 </interactionModule>
 [...]
</platformModule>
```

Listing B.27 – Définitions des connecteurs pour l'envoi de message authentifié par socket.

Le connecteur de donnée partagée avec chiffrement est défini selon le même principe.

Annexe C

Documentation sur MoBACT

C.1 Installation de MoBACT

L'utilisation de MoBACT requiert l'installation de TSNsched [SSN19] dont le dépôt est à l'adresse suivante : https://github.com/ACassimiro/TSNsched. TSNsched nécessite à son tour l'installation de Z3 [dMB08] dont le dépôt est à l'adresse suivante : https://github.com/Z3Prover/z3. L'installation de MoBACT nécessite également l'installation de Java Ant (https://ant.apache.org/).

Z3 est disponible dans les dépôts de la plupart des distributions Linux, par exemple avec Ubuntu :

```
# sudo apt install z3 libz3-java
```

Pour l'installation de TSNsched, utiliser la commande suivante en dehors du dossier contenant l'installation de MoBACT :

```
# git clone https://github.com/ACassimiro/TSNsched.git
```

La compilation de MoBACT avec Ant se fait avec la commande suivante (en ajoutant TSNsched.jar au *classpath* :

```
# ant -lib <chemin vers TSNsched>/libs build
```

Le résultat de la compilation est le fichier `mobact.jar`.

Enifn, pour pouvoir exploiter les résultats de MoBACT, l'installation d'un simulateur réseau est nécessaire, par exemple NeSTiNg dont le dépôt et les instructions d'installation sont disponible à l'adresse suivante : https://gitlab.com/ipvs/nesting.

C.2 Exécution de MoBACT

L'exécution de MoBACT se fait avec la commande suivante :

```
# java -Djava.library.path="<chemin vers>/libz3java.so" -jar mobact.jar
```

Les différents arguments qu'il est possible de passer à MoBACT sont les suivants :

— le chemins vers les fichiers contenant le modèle d'entrée au format XML :

— les cibles vers lesquelles la génération de modèles de simulation et d'analyse doit être faite, `-nesting`, `-pegase` et `-mininet` sont disponibles, l'option `-all` permet de cibler l'ensemble de ces outils ;

— l'option `-tas` permet d'activer la synthèse de configuration pour le TAS ;

— le nom de l'environnement racine du modèle, celui qui référence les autres environnement présents dans le modèle d'entrer.

Par exemple, pour exécuter MoBACT sur un modèle appelé CPS4EU, la commande est la suivante :

```
# java -Djava.library.path="/usr/lib/x86_64-linux-gnu/jni/" -jar mobact.jar cps4eu/*.xml -all -tas ::cps4eu_full
```

elle permet d'exécuter MoBACT sur le modèle d'entrée dont l'environnement racine est appelé `::cps4eu_full`, avec la synthèse de configuration du TAS et en générant les modèles de simulation et d'analyse pour l'ensemble des cibles disponibles.

Cette exécution produira un dossier appelé `cps4eu_full_output` qui contiendra la documentation, un graphe de la topologie, un fichier de log contenant le résultat de la génération de configuration et l'ensemble des modèles d'analyse et de simulation.

C.3 Utilisation typique de MoBACT

MoBACT a besoin d'un modèle d'entrée contenant les informations suivantes :

— la topologie du réseau ;

— le modèle des flux de données ;

— les exigences du systèmes.

De manière générale, le modèle de l'architecture logicielle est un fichier nommé `ucm_models/model.ucm` qui contient la déclaration des composants, leur déploiement et le modèle de la topologie du réseau. MoBACT ne prenant comme entrée que des fichiers au format XML, il est nécessaire de généré un fichier dans ce format à partir du modèle dans le format UCM. L'outil Sigil-UCM [Ver23] permet de le faire grâce à un clique droit sur le fichier `ucm_models/model.ucm` puis en sélectionnant `Sigil-UCM` → `Export to XML, processing the complete definitions`. Le résultat de cette génération est le fichier `gen_xml/model.xml`.

La génération du modèle des flux de données s'effectue de la même façon, par un clique droit sur le fichier contenant le modèle de l'architecture logicielle puis en sélectionnant `Sigil-UCM` → `Generate all, Pharos platform`. Cette action permet de générer le code technique de l'application, des modèles d'analyse de cette application et le modèle des flux de données, disponible dans les fichiers présents dans le dossier `gen_exec_analysis/toml`.

Il est ensuite nécessaire de convertir ces fichiers au format TOML dans le format XML de Sigil-UCM. Cette étape est effectuée par l'utilisation du script `ts2x.py` disponible dans le dossier `tools` en invoquant la commande suivante :

```
# python3 tools/ts2x.py gen_exec_analysis/toml/*.toml
```

qui créera un fichier au format XML pour chaque fichier au format TOML.

Il est possible de vérifier le modèle des flux de données en l'important dans l'outil Sigil-UCM en effectuant un clique droit sur le dossier `streams` puis en sélectionnant `Sigil-UCM` → `Parse the UCM project file` sur chaque fichier dont l'extension est `.ucmproject`.

Il est maintenant possible d'exécuter MoBACT en regroupant l'ensemble des fichiers au format XML composant le modèle d'entrée : la topologie du système, le modèle des flux de données et les exigences du système. Un script est disponible pour lancer facilement cette exécution avec la commande suivante :

```
# tools/gen_mobact.sh <chemin vers>/mobact.jar
```

Glossaire

AADL : Architecture Analysis and Design Language (précédemment Avionics Architecture Description Language).
CBS : Credit-Based Shaper.
CBSE : Component-Based Software Engineering.
CCM : CORBA Component Model.
CORBA : Common Object Request Broker Architecture.
EMF : Eclipse Modeling Framework.
IDL : Interface Definition Language.
GCL : Gate Control List.
gPTP : Generalized Precision Time Protocol.
OMG : Object Management Group.
SDN : Software-Defined Network.
SMT : Satisfiability Modulo Theories.
TAS : Time Aware Shaper.
TSN : Time-Sensitive Networking.
UCM : Unified Component Model.
UDP : User Datagram Protocol.
UML : Unified Modeling Language.
XML : eXtensible Markup Language.

Bibliographie

[15819] Ieee standard for a precision clock synchronization protocol for networked measurement and control systems, 2019.

[80222] IEEE Standard for Ethernet, 2022.

[AD94] Rajeev Alur and David L. Dill. A theory of timed automata. *Theoretical Computer Science*, 126(2) :183–235, 1994.

[ARI01] ARINC 429 : Digital Information Transfer Systems (DITS), 2001.

[ARI09] ARINC 664p7 : Aircraft Data Network, part 7, Avionics Full-Duplex Switched Ethernet Network, 2009.

[AS20] IEEE 802.1AS 2020 – Timing and Synchronization for Time-Sensitive Applications, 2020.

[BA21] IEEE Standard for Local and Metropolitan Area Networks – Audio Video Bridging (AVB) Systems, 2021.

[BBRR18] Nassima Bennammar, Henri Bauer, Frédéric Ridouard, and Pascal Richard. Timing Analysis of AVB Ethernet network using the Forward end-to-end Delay Analysis. In *26th International Conference on Real-Time Networks and Systems (RTNS)*, pages 223–233, 2018.

[BD19] Marc Boyer and Hugo Daigmorte. Impact on credit freeze before gate closing in cbs and gcl integration into tsn. In *Proceedings of the 27th International Conference on Real-Time Networks and Systems*, RTNS '19, page 80–89, New York, NY, USA, 2019. Association for Computing Machinery.

[BLL+96] Johan Bengtsson, Kim Larsen, Fredrik Larsson, Paul Pettersson, and Wang Yi. UPPAAL – a tool suite for automatic verification of real-time systems. In *Hybrid Systems III*, 1996.

[br16] Ieee standard for ethernet amendment 5 : Specification and management parameters for interspersing express traffic, 2016.

[CAC+18] Jingyue Cao, Mohammad Ashjaei, Pieter J. L. Cuijpers, Reinder J. Bril, and Johan J. Lukkien. An independent yet efficient analysis of bandwidth reservation for credit-based shaping. In *2018 14th IEEE International Workshop on Factory Communication Systems (WFCS)*, pages 1–10, 2018.

[CAN11] CAN with Flexible Data-Rate, Version 1.1, 2011.

[CAN15] Road vehicles – Controller Area Network (CAN) – Data link layer and physical signaling, 2015.

[CB17] Frame Replication and Elimination for Reliability, 2017.

[CBO13] Concise Binary Object Representation (CBOR). https://www.rfc-editor.org/rfc/rfc8949.html, 2013.

[CCBL16a] Jingyue Cao, Pieter J. L. Cuijpers, Reinder J. Bril, and Johan J. Lukkien. Tight worst-case response-time analysis for ethernet avb using eligible intervals. In *2016 IEEE World Conference on Factory Communication Systems (WFCS)*, pages 1–8, 2016.

[CCBL16b] Jingyue Cao, Pieter J.L. Cuijpers, Reinder J. Bril, and Johan J. Lukkien. Independent yet tight wcrt analysis for individual priority classes in ethernet avb. In *Proceedings of the 24th International Conference on Real-Time Networks and Systems*, RTNS '16, page 55–64, New York, NY, USA, 2016. Association for Computing Machinery.

[CCBL18] Jingyue Cao, Pieter J. Cuijpers, Reinder J. Bril, and Johan J. Lukkien. Independent wcrt analysis for individual priority classes in ethernet avb. *Real-Time Systems*, 54(4) :861–911, oct 2018.

[CCM06] CORBA Component Model (CCM). https://www.omg.org/spec/CCM, 2006.

[COCS16] Silviu S. Craciunas, Ramon Serna Oliver, Martin Chmelík, and Wilfried Steiner. Scheduling Real-Time Communication in IEEE 802.1Qbv Time Sensitive Networks. In *24th International Conference on Real-Time Networks and Systems (RTNS)*, 2016.

[COR21] Common Object Request Broker Architecture (CORBA). https://www.omg.org/spec/CORBA, 2021.

[DAB14a] Joan Adrià Ruiz De Azua and Marc Boyer. Complete modelling of avb in network calculus framework. In *Proceedings of the 22nd International Conference on Real-Time Networks and Systems*, RTNS '14, page 55–64, New York, NY, USA, 2014. Association for Computing Machinery.

[DAB14b] Joan Adrià Ruiz De Azua and Marc Boyer. Complete modelling of avb in network calculus framework. In *Proceedings of the 22nd International Conference on Real-Time Networks and Systems*, RTNS '14, page 55–64, New York, NY, USA, 2014. Association for Computing Machinery.

[Dan06] Deployment and Configuration of component-based Distributed Applications. https://www.omg.org/spec/DEPL, 2006.

[dMB08] Leonardo de Moura and Nikolaj Bjørner. Z3 : An Efficient SMT Solver. In C. R. Ramakrishnan and Jakob Rehof, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340. Springer Berlin Heidelberg, 2008.

[Doc21] Théo Docquier. *Méthodologies pour l'évaluation de performances d'architectures réseaux smart grids*. PhD thesis, Université de Lorraine, 2021. Thèse de doctorat dirigée par Song, Ye-Qiong et Chevrier, Vincent Informatique Université de Lorraine 2021.

[EMF] Eclipse Modeling Framework. https://eclipse.dev/modeling/emf/.

[FFG06] F. Francés, Christian Fraboul, and J Grieu. Using Network Calculus to optimize the AFDX network. In *Conference ERTS'06*, Toulouse, France, January 2006.

[FHC+19] Jonathan Falk, David Hellmanns, Ben Carabelli, Naresh Nayak, Frank Durr, Stephan Kehrer, and Kurt Rothermel. NeSTiNg : Simulating IEEE Time-Sensitive Networking (TSN) in OMNeT++. In *International Conference on Networked Systems (NetSys)*, 2019.

[fie23] IEC 61158-1 :2023 Industrial communication networks - Fieldbus specifications - Part 1 : Overview and guidance for the IEC 61158 and IEC 61784 series, 2023.

[FMHH01] Thomas Fuehrer, Bernd Mueller, Florian Hartwich, and Robert Hugel. Time triggered can (ttcan). *SAE Transactions*, 110 :143–149, 2001.

[Ful09] John Nels Fuller. Calculating the delay added by qav stream queue. https://grouper.ieee.org/groups/802/1/files/public/docs2009/av-fuller-queue-delay-calculation-0809.pdf, 2009.

[GHS+21] Wang Guo, Yanhong Huang, Jianqi Shi, Zhe Hou, and Yang Yang. A Formal Method for Evaluating the Performance of TSN Traffic Shapers using UPPAAL. In *IEEE 46th Conference on Local Computer Networks (LCN)*, 2021.

[HBA+21] Bahar Houtan, Albert Bergström, Mohammad Ashjaei, Masoud Daneshtalab, Mikael Sjödin, and Saad Mubeen. An Automated Configuration Framework for TSN Networks. In *IEEE 22nd International Conference on Industrial Technology (ICIT)*, 2021.

[HGO16] Peter Heise, Fabien Geyer, and Roman Obermaisser. TSimNet : An Industrial Time Sensitive Networking Simulation Framework Based on OMNeT++. In *IFIP 8th International Conference on New Technologies, Mobility and Security (NTMS)*, 2016.

[IDL18] Interface Definition Language (IDL). https://www.omg.org/spec/IDL, 2018.

[IP81] RFC 791 – Internet Protocol, 1981.

[jum06] Ieee standard for information technology– telecommunications and information exchange between systems– local and metropolitan area networks– specific requirements part 3 : Carrier sense multiple access with collision detection (csma/cd) access method and physical layer specifications, 2006.

[Kop11] Hermann Kopetz. *Real-Time Systems - Design Principles for Distributed Embedded Applications*. Real-Time Systems Series. Springer, 2011.

[LHM10] Bob Lantz, Brandon Heller, and Nick McKeown. A network in a laptop : rapid prototyping for software-defined networks. In *9th ACM SIGCOMM Workshop on Hot Topics in Networks (Hotnets)*, 2010.

[LL73] C. L. Liu and James W. Layland. Scheduling algorithms for multiprogramming in a hard-real-time environment. *J. ACM*, 20 :46–61, jan 1973.

[LZW+21] Jin Lv, Yongxin Zhao, Xi Wu, Yongjian Li, and Qiang Wang. Formal Analysis of TSN Scheduler for Real-Time Communications. *IEEE Transactions on Reliability*, 2021.

[M.10] Bjorklund M. YANG - A Data Modeling Language for the Network Configuration Protocol (NETCONF), 2010.

[MAB+08] Nick McKeown, Tom Anderson, Hari Balakrishnan, Guru Parulkar, Larry Peterson, Jennifer Rexford, Scott Shenker, and Jonathan Turner. OpenFlow : Enabling Innovation in Campus Networks. *SIGCOMM Computer Communication Review*, 38(2) :69–74, 2008.

[MAC18] IEEE 802.3 – IEEE Standard for Ethernet, 2018.

[MAR19] UML Profile for MARTE : Modeling and Analysis of Real-Time Embedded Systems. http://www.omg.org/spec/MARTE, 2019.

[MIL18] MIL-STD-1553B : Digital Time Division Command/Response Multiplex Data Bus, 2018.

[MS17] Dorin Maxim and Ye-Qiong Song. Delay analysis of AVB traffic in time-sensitive networks (TSN). In *Proceedings of the 25th International Conference on Real-Time Networks and Systems*, pages 18–27, Grenoble France, October 2017. ACM.

[MVG+23] Lisa Maile, Dominik Voitlein, Alexej Grigorjew, Kai-Steffen J. Hielscher, and Reinhard German. On the validity of credit-based shaper delay guarantees in decentralized reservation protocols. In *Proceedings of the 31st International Conference on Real-Time Networks and Systems*, RTNS '23, page 108–118, New York, NY, USA, 2023. Association for Computing Machinery.

[NDR16] Naresh Ganesh Nayak, Franck Dürr, and Kurt Rothermel. Time-Sensitive Software-Defined Network (TSSDN) for Real-time Applications. In *24th International Conference on Real-Time Networks and Systems (RTNS)*, 2016.

[OSI94] ISO/IEC 7498-1 :1994 Information Technology – Open System Interconnection – Basic Reference Model : The Basic Model, 1994.

[Pap17] Papyrus Software Designer. https://marketplace.eclipse.org/content/papyrus-software-designer, 2017.

[PO18] Maryam Pahlevan and Roman Obermaisser. Genetic Algorithm for Scheduling Time-Triggered Traffic in Time-Sensitive Networks. In *IEEE 23rd International Conference on Emerging Technologies and Factory Automation (ETFA)*, 2018.

[Puj08] Guy Pujolle. *Les Réseaux*. Eyrolles, 2008.

[Q18] IEEE 802.1Q 2018 – Bridges and Bridged Networks, 2018.

[Qat10] Ieee 802.1Qat : Stream Reservation Protocol (SRP), 2010.

[Qav16] IEEE 802.1Qav : Forwarding and Queuing for Time-Sensitive Streams, 2016.

[Qbu16] Ieee standard for local and metropolitan area networks – bridges and bridged networks – amendment 26 : Frame preemption, 2016.

[Qbv16] IEEE 802.1Qbv : Enhancements for Scheduled Traffic, 2016.

[RMJA11] Enns R., Bjorklund M., Schoenwaelder J., and Bierman A. Network Configuration Protocol (NETCONF), 2011.

[RSA78] R. L. Rivest, A. Shamir, and L. Adleman. A Method for Obtaining Digital Signatures and Public-Key Cryptosystems. *Commun. ACM*, 21(2) :120–126, feb 1978.

[SBH+09] Wilfried Steiner, Günther Bauer, Brendan Hall, Michael Paulitsch, and Srivatsan Varadarajan. TTEthernet Dataflow Concept. In *IEEE 8th International Symposium on Network Computing and Applications*, 2009.

[SSN19] Aellison Cassimiro T. dos Santos, Ben Schneider, and Vivek Nigam. TSNsched : Automated Schedule Generation for Time Sensitive Networking. In *Formal Methods in Computer Aided Design (FMCAD)*, 2019.

[SV06] Thomas Stahl and Markus Völter. *Model-Driven Software Development*. John Wiley & Sons, Ltd, 2006.

[SV08] Teresa Schuster and Dinesh Verma. Networking concepts comparison for avionics architecture. In *2008 IEEE/AIAA 27th Digital Avionics Systems Conference*, 2008.

[SV19] Maxime Samson and Thomas Vergnaud. Automatic Generation of Test Oracles from Component Based Software Architectures. In *Testing Software and Systems*, pages 261–269. Springer International Publishing, 2019.

[SVD+21] Maxime Samson, Thomas Vergnaud, Éric Dujardin, Laurent Ciarletta, and Ye-Qiong Song. Une Approche de Génération Automatique de Configuration Basée sur les Modèles pour les Réseaux TSN. In *ETR2021 - L'École d'Été Temps Réel 2021*, Poitiers, France, 2021. LIAS-ISAE/ENSMA.

[SVD+22] Maxime Samson, Thomas Vergnaud, Éric Dujardin, Laurent Ciarletta, and Ye-Qiong Song. A Model-Based Approach to Automatic Generation of TSN Network Simulations. In *2022 IEEE 18th International Conference on Factory Communication Systems (WFCS)*, pages 1–8, 2022.

[SVD+23] Maxime Samson, Thomas Vergnaud, Éric Dujardin, Laurent Ciarletta, and Ye-Qiong Song. Computing Data Streams in Real-Time Networks from Component-Based Software Engineering. In *IEEE 28th International Conference on Emerging Technologies and Factory Automation (ETFA)*, 2023.

[TCP81] Transmission Control Protocol, 1981.

[UCM21] Unified Component Model for Distributed, Real-Time and Embedded Systems (UCM). http://www.omg.org/spec/UCM, 2021.

[UDP80] User Datagram Protocol, 1980.

[UML17] Unified Modeling Language (UML). https://www.omg.org/spec/UML, 2017.

[Ver23] Thomas Vergnaud. Documentation of Sigil-UCM. Technical Report TRT-Fr/STI/LISL/TVE,230005, Thales Research & Technology, September 2023.

[Ves07] Steve Vestal. Preemptive scheduling of multi-criticality systems with varying degrees of execution time assurance. In *28th IEEE International Real-Time Systems Symposium (RTSS 2007)*, pages 239–243, 2007.

[WEE+08] Reinhard Wilhelm, Jakob Engblom, Andreas Ermedahl, Niklas Holsti, Stephan Thesing, David Whalley, Guillem Bernat, Christian Ferdinand, Reinhold Heckmann, Tulika Mitra, Frank Mueller, Isabelle Puaut, Peter Puschner, Jan Staschulat, and Per Stenström. The worst-case execution-time problem—overview of methods and survey of tools. *ACM Trans. Embed. Comput. Syst.*, 7(3), may 2008.

[ZPZ+18] Luxi Zhao, Paul Pop, Zhong Zheng, Hugo Daigmorte, and Marc Boyer. Improving worst-case end-to-end delay analysis of multiple classes of AVB traffic in TSN networks using network calculus. In *DTU Compute Technical Report*, 2018.

[ZPZ+21] Luxi Zhao, Paul Pop, Zhong Zheng, Hugo Daigmorte, and Marc Boyer. Latency Analysis of Multiple Classes of AVB Traffic in TSN with Standard Credit Behavior Using Network Calculus. *IEEE Transactions on Industrial Electronics*, 68(10) :10291–10302, 2021.

[ZPZL18] Luxi Zhao, Paul Pop, Zhong Zheng, and Qiao Li. Timing analysis of avb traffic in tsn networks using network calculus. In *Proceedings of 2018 IEEE Real-Time and Embedded Technology and Applications Symposium*, pages 25–36, United States, 2018. IEEE. 2018 IEEE Real-Time and Embedded Technology and Applications Symposium, RTAS 2018 ; Conference date : 11-04-2018 Through 13-04-2018.

Résumé

Nous étudions le processus de conception et de configuration de réseaux temps réel utilisant les standards *Time-Sensitive Networking* (TSN). Le groupe de travail IEEE 802.1 TSN a publié un ensemble de standards ajoutant de nouvelles fonctionnalités aux normes utilisés par les réseaux Ethernet commutés. L'objectif de ces nouvelles fonctionnalités est de permettre la mise en place de réseaux Ethernet déterministes, ce qui rend possible leur utilisation dans le cadre d'applications temps réels.

L'obtention de cette propriété de déterminisme a néanmoins un coût : l'augmentation de la complexité dans la conception et dans la configuration de ces réseaux. Ces nouvelles fonctionnalités entraînent une augmentation de l'effort de configuration destiné à déployer un réseau capable de respecter ses exigences temps réel. De plus, le processus de conception de ces réseaux faisant usage d'outils de conception tel que des simulateurs réseau, cette augmentation de complexité se répercute également sur ces derniers.

Nous proposons dans cette thèse une approche outillée d'assistance à la conception de réseau TSN qui repose sur la modélisation du réseau, des applications qui l'utiliseront et la génération automatique. Nous proposons d'abord une approche de modélisation pour les réseaux TSN. Nous la lions ensuite à une approche de modélisation logicielle afin de mettre en place un processus de génération automatique qui complète le modèle du réseau avec les flux de données. À l'aide de ce modèle, nous proposons une méthode de calcul de la configuration du *Credit-Based Shaper* et du *Time Aware Shaper*. Enfin, nous avons développé un outil générant un ensemble de modèles de simulation à destination de plusieurs simulateurs réseau ainsi que les fichiers de configuration des équipements réseau.

Mots-clés: réseau, temps réel, TSN, MBSE, UCM , configuration, simulation.

Abstract

We are studying the design and configuration process of real-time networks that use the Time-Sensitive Networking (TSN) standards. The IEEE 802.1 TSN working group published a set of standards which adds multiple new functionalities to the switched Ethernet standards. The goal of these new functionalities is to allow the design of deterministic Ethernet networks, which makes their use possible in the context of real-time applications.

Making it possible to design deterministic Ethernet networks has a cost : the increase in design and configuration complexity of the network. These new functionalities makes the configuration effort needed to guarantee the respect of real-time constraints more important. Moreover, since the design process of real-time networks makes use of design tools such as network simulators, this increase in complexity also has an impact on them.

In this thesis, we propose a tooled approach to assist the design of TSN networks which relies on network and software modeling and automatic generation. We first propose a network modeling approach for TSN networks. We then link it with a software modeling approach in order to automatically enrich the model of the network with a model of the data streams. Using the data contained in this model, we propose a method to compute the configuration of the Credit-Based Shaper and Time Aware Shaper. Finally, we developed a tool which produces a set of simulation models for different network simulators aswell as configuration files for network equipment.

Keywords: network, real-time, TSN, MBSE, UCM, configuration, simulation.