Application Driven Networking in Dynamic Environments Cas d'application simulation distribuée : implémentation, validation et évaluation

Slim Abdellatif, Armel Francklin Simo Tegueu, Pascal Berthou, Thierry Villemur

Projet ANR- 13-ASTR-0024 ADN

Projet ANR- 13-ASTR-0024

Application Driven Networking in Dynamic Environments

Livrable 2.1

Cas d'application simulation distribuée : implémentation, validation et évaluation

Référence document	ANR-13-ASTR-0024-L2.1
Liste des auteurs	S. Abdellatif, F. Simo Tegue, M. Capelle, P. Berthou, T. Villemur, M.J. Huguet *LAAS-CNRS, Université de Toulouse, CNRS, INSA, UT2J, UPS, Toulouse, France*
Version	1.0
Date de livraison	30/03/2016

TABLE DES MATIERES

1 Introduction 5

2 Architecture ADN 6
2.1 Rappel des principes clefs du concept ADN 6
2.2 Architecture de référence ADN 7
2.2.1 Request handler 8
2.2.2 Flow Aggregator 8
2.2.3 Virtual Network Resource Allocator 9
2.2.4 Virtual Network Deployer 9
2.2.5 Application Classifier 9
2.2.6 Autonomic Manager 9
2.3 Synthèse 10

3 Prototype ADN 11
3.1 Implémentation vue à l'échelle des composants et dynamique du système. 11
3.2 Implémentation vue à l'échelle de chaque composant 13
3.2.1 Application Classifier 13
3.2.1.1 Capture des besoins de l'application 15
3.2.1.1 Traduction des besoins de l'application en service réseau 18
3.2.2 VNET Deployer 18
3.2.3 VNET Resource Allocator 21
3.2.4 Request Handler 22

4 Démonstrateur ADN 23
4.1 Introduction 23
4.2 Application dynamique DDS considérée 24
4.2.1 Description des topics 24
4.2.2 Spécification des QoS des topics 26
4.2.3 Composante dynamique de l'application considérée 27
4.3 Infrastructure réseau considérée 28
4.4 Scénario applicatif et résultats obtenus 29
4.4.1 Description du scénario applicatif 29
4.4.2 Résultats du démonstrateur 33
4.5 Conclusion 35

5 Evaluations 36
5.1 Introduction 36
5.2 Modèle de simulation considéré 36
5.2.1 Modèles de réseau 36
5.2.2 Modèles de charge 38
5.2.3 Les paramètres d'expérimentation 39
5.2.4 Fonctionnement des expérimentations 39
5.3 Métriques de performance 40
5.3.1 Le taux d'acceptation des requêtes 40
5.3.2 Les revenus générés 40
5.3.3 La charge globale des liens du réseau 41
5.3.4 Le taux d'utilisation des liens 41
5.4 Résultats expérimentaux sur le réseau Géant 42
5.4.1 Comparaison avec des méthodes heuristiques 42
5.4.2 Caractérisation des temps de calcul 42
5.4.3 Validation de la durée de simulation 44
5.4.4 Influence des paramètres du modèle 45
5.4.5 Visualisation de l'équilibre de l'utilisation des liens dans le réseau 49
5.4.6 Analyse du taux d'acceptation des instances de type unicast et multicast 51
5.5 Résultats expérimentaux sur le réseau de Campus 52

5.5.1 Comparaison avec des méthodes heuristiques 52
5.5.2 Caractérisation des temps de calcul 53
5.5.3 Taux d'utilisation des liens et des noeuds 53
6 Conclusion 55
7 References 56
Annexe A : Définition des topics de l'application considérée 58

LISTE DES FIGURES

Figure 1 – Fonction de provisionnement du service ADN 7
Figure 2 – Architecture fonctionnelle du réseau ADN 7
Figure 3 – Fonctionnement du « Request Handler » lors d'une demande d'établissement d'un VNET . 8
Figure 4 – Synthèse de l'approche ADN 10
Figure 5 - Diagramme de composants du système ADN 11
Figure 6 - Description fonctionnelle des interfaces 12
Figure 7 - Diagramme de séquence du cas d'utilisation « Install VNET » 12
Figure 8 - Diagramme de séquence du cas d'utilisation « Uninstall VNET » 12
Figure 9 – Package de base et diagramme des classes du service ADN 13
Figure 10 - Structure interne niveau package du composant Application Classifier 14
Figure 11 - Structure interne du package « fr.laas.projects.adn.applicationClassifier.ddsTypes » 14
Figure 12 - Structure interne du package « fr.laas.projects.adn.applicationClassifier.AppReqCapture » 15
Figure 13 - Relation entre un builtin topic, ses instances et leurs échantillons 17
Figure 14 - diagramme d'état d'une instance et de sa vue 18
Figure 15 - Architecture du contrôleur réseau Floodlight 19
Figure 16 - Structure interne niveau package du composant VNET Deployer 20
Figure 17 - Structure interne du package « fr.laas.projects.adn.vnetdeployer » 21
Figure 18 - Diagramme des classes de Concert technology et de CPLEX 21
Figure 19 - Structure interne niveau package du composant Request Handler 22
Figure 20 – Application de simulation considérée 23
Figure 21 – Topics DDS relatifs à l'application considérée 24
Figure 22 – Topologie du réseau du LAAS 28
Figure 23 – Illustration du scénario à l'instant t0 29
Figure 24 – Illustration du scénario à l'instant t1 31
Figure 25 – Illustration du scénario à l'instant t2 32
Figure 26 – Illustration du scénario à l'instant t3 32
Figure 27 – Illustration du scénario à l'instant t4 33
Figure 28 – Illustration du scénario à l'instant final t5 33
Figure 29 – Résultats des allocations suite aux requêtes VNET soumises à l'instant t0 34
Figure 30 – Résultats des allocations suite aux requêtes VNET soumises à l'instant t1 34
Figure 31- Chemins des données des liens virtuels supportant les échanges relatifs au topic «*EtatsVehiculeSim* » calculés à l'instant t1 35

LISTE DES TABLEAUX

Tableau 1 - Structure de données associées aux Builtin Topics 17
Tableau 2 - Description des services exposées via REST 18
Tableau 2 - Topics DDS de l'application de simulation distribuée considérée 27
Tableau 3 – caractérisation de la dynamique de l'application 28
Tableau 4 – Liste des flux de données du démonstrateur à l'instant t_0 30
Tableau 5 – Liste des liens agrégés de la requête VNET soumise à l'instant t_0 au « Resource Allocator » 30
Tableau 6 – Liste des requêtes issues du Simulateur S4 à l'instant t_1 31
Tableau 7 – Liste des liens agrégés de la requête VNET soumise à l'instant t_1 au « Resource Allocator » pour supporter les données produites par S4 31

1 INTRODUCTION

La tâche 2 du projet ADN porte sur l'application de l'approche ADN à un premier cas d'étude qui utilise une application de simulation distribuée au travers d'un réseau filaire. Cette application implique plusieurs simulateurs de véhicule dirigés par des apprenants qui participent à un exercice ou une formation à la conduite en groupe. Des postes instructeurs recueillent les données de certains simulateurs et les renvoient à l'écran pour le compte d'instructeurs en charge de superviser la formation. Enfin, un serveur de simulation récupère et archive l'ensemble des données des simulateurs pour un éventuel rejeu de la simulation. L'application considérée repose sur l'intergiciel DDS (Data Distribution Service) de l'OMG (Object Management Group) pour distribuer les flux de données applicatives entre les différents composants de l'application. Le middleware DDS repose, à son tour, sur le réseau de type ADN (Application Driven Network) défini dans [1] pour déployer à la volée les services réseau permettant de supporter la distribution DDS des données.

Les objectifs de ce livrable sont les suivants : Décrire le prototype ADN qui a été implémenté à travers la description des diagrammes de conception de chacun de ses composants ; Présenter la plateforme de démonstration qui a été mise en place, à savoir l'application et l'infrastructure réseau retenues ainsi que le scénario applicatif qui a été choisi pour la démonstration ; Enfin, présenter les principaux résultats des analyses de performances qui ont principalement porté sur l'algorithme d'allocation de ressources.

Le plan de ce rapport se présente comme suit. La section 2 décrit très brièvement l'architecture ADN qui a été prototypée. Cette dernière présente, par rapport à l'architecture préliminaire décrite dans le livrable précédent [1], quelques petites évolutions qui ouvrent son champ d'application. La section 3 décrit l'implémentation du prototype ADN. La section 4 décrit la plateforme de démonstration qui a été mise en place et son utilisation pour un scénario applicatif donné. La section 5 décrit les analyses de performances de l'algorithme d'allocation de ressources décrit dans [1] et qui ont porté sur deux topologies réseau réelles, l'une issue du réseau Géant2 et l'autre de l'infrastructure réseau de campus de notre laboratoire d'appartenance. La dernière section conclut le livrable.

2 ARCHITECTURE ADN

Nous présentons dans ce qui suit l'architecture fonctionnelle d'un réseau ADN bâti sur une infrastructure réseau de type SDN. L'architecture proposée se veut plus générale que celle introduite dans le livrable L1.2 du projet [1] dans la mesure où les services ADN qu'elle offre ne ciblent pas exclusivement les applications DDS mais peuvent concerner d'autres types d'application. La première section rappelle et complète les principes du réseau ADN que l'on propose dans le cadre de ce projet. La section suivante décline l'architecture ADN.

2.1 Rappel des principes clefs du concept ADN

L'idée centrale d'un réseau ADN est de fournir des services (i.e. des chemins de données) personnalisées aux applications sur la base d'une description fine des flux applicatifs échangés et de leurs besoins en terme de Qualité de Service (QdS) (que l'on appellera dans la suite : le profil de communication de l'application). Les motivations d'un tel choix sont qu'une connaissance précise des besoins applicatifs est une condition nécessaire pour déployer le « bon » service (celui qui respecte avec précision les besoins) avec une utilisation optimale des ressources. En comparaison aux travaux antérieurs [2][3][4][5][6][7][8][9] visant la fourniture de services avec un certain niveau de QdS, c'est l'une des caractéristiques clef de l'approche ADN que nous proposons. Il est à noter que cette connaissance fine du profil de communication d'une application n'est pas une condition sine qua non pour l'utilisation des services ADN, comme expliqué ci-avant.
Le profil de communication d'une application peut être établi par le réseau ADN à partir d'une description explicite de ses attentes et besoins. Au besoin, le réseau ADN peut compléter ce profil par le déploiement sur le réseau de mécanismes d'estimation du trafic applicatif. Pour les applications DDS qui sont au centre de ce projet, la description du profil de communication d'une application va jusqu'à l'identification des flux de données applicatives (définies par les topics DDS [13]) et des différents paramètres de QdS associés à leur distribution. Ces profils peuvent être soumis explicitement par le middleware (ce qui à notre connaissance n'est pas dans les versions du middleware DDS du marché) ou par un agent applicatif externe avec potentiellement un administrateur dans la boucle ou tout simplement non communiqué explicitement au réseau ADN. Dans ce dernier cas, des mécanismes d'inspection profonde de paquets (que l'on notera DPI pour Deep Packet Inspection) alimenté avec les signatures applicatives pertinentes sont déployées aux extrémités du réseau pour classer le trafic appartenant aux applications supposées utiliser le service ADN et déduire leurs besoins en QoS. Pour les applications qui n'explicitent pas leur besoins (à l'inverse des applications DDS considérées dans ce projet), des techniques d'estimation statistique du tarfic peuvent être utilisées pour évaluer les besoins de l'application. Il est clair que, dans ce cas de figure, la précision et l'exactitude du profil de communication sont impactés.

Un autre principe est que le service réseau fourni aux applications est exprimé sous la forme d'un réseau virtuel (que l'on notre VNET pour Virtual NETwork) composé d'un ensemble de liens virtuels de bout-en-bout (de hôte à hôte). Chaque lien virtuel est soit de type point-à-point, soit, point-multipoints, et est caractérisé par une exigence de débit et un délai de transfert maximal à respecter.

Le réseau ADN que nous proposons repose sur une infrastructure réseau de type SDN/Openflow [12]. Cette dernière peut être entièrement ou partiellement dédiée au réseau ADN. Dans ce dernier cas, une technique de virtualisation réseau [14] est appliquée à l'infrastructure physique pour affecter exclusivement des partitions (ou slices) des éléments du réseau à l'infrastructure

SDN/openflow virtuelle sur laquelle repose le réseau ADN. Comme illustré par la Figure 1, une application SDN de contrôle réseau implémente l'approche ADN et est en charge de fournir les services ADN suscités. Cette application réseau repose sur une interface nord de bas-niveau qui ne fait que reprendre la syntaxe et les possibilités du protocole Openflow.

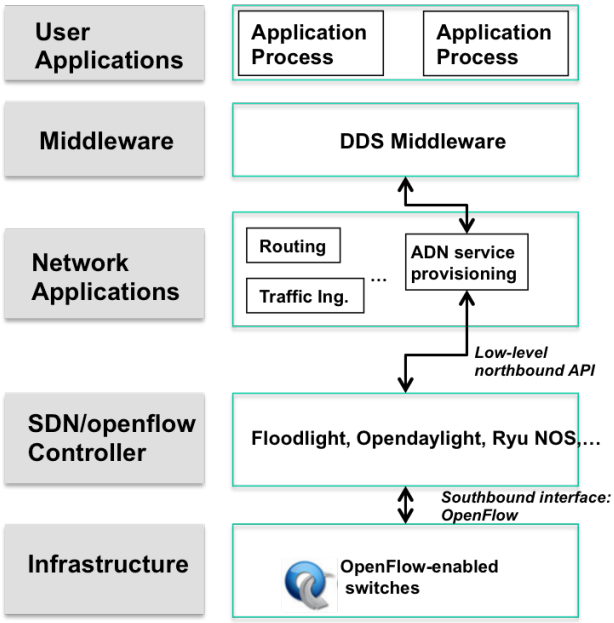


Figure 1 – Fonction de provisionnement du service ADN

Le dernier principe de conception est de construire un réseau ADN autonomique. Nous ciblons les propriétés d'auto-configuration, d'auto-protection et d'auto-optimisation.

2.2 Architecture de référence ADN

La **Figure 2** décrit l'architecture fonctionnelle de l'application de contrôle de réseau « ADN service Provisionning » qui implémente l'approche ADN que nous proposons. Une brève présentation des différents composants est présentée ci-après.

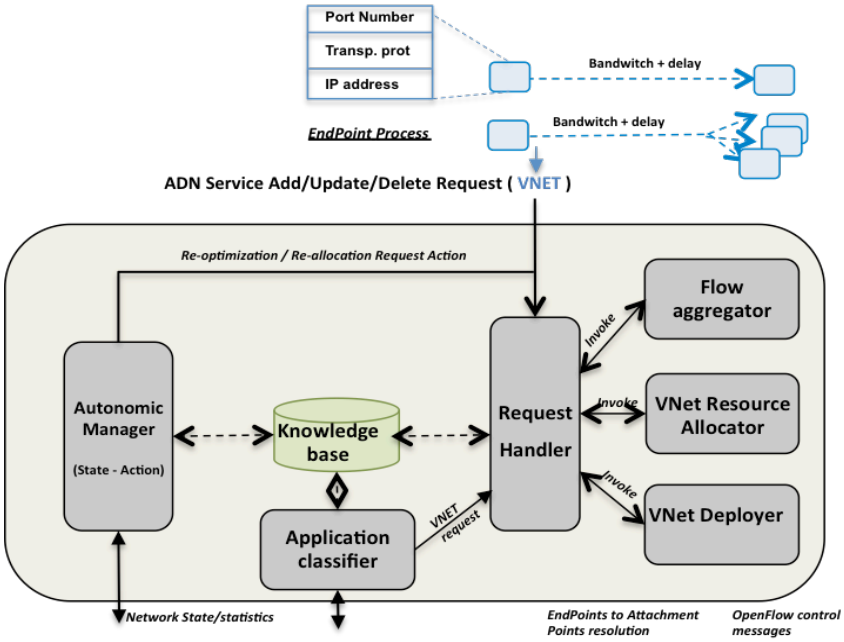


Figure 2 – Architecture fonctionnelle du réseau ADN

2.2.1 REQUEST HANDLER

Ce composant joue le rôle de « front-end » de l'application de contrôle et orchestre l'exécution des différents composants impliqués du traitement d'une requête de rajout, de mise à jour ou de retrait d'un VNET. A la réception d'une requête d'ajout de VNET, il lance l'exécuteur du composant « flow aggregator » pour vérifier s'il est possible d'agréger ou de regrouper certains flux du VNET. Ensuite, il lance le composant « Resource Allocator » pour calculer les chemins de données supportant le VNET avec les ressources à réserver le long de ces chemins. Enfin, il lance le « VNET Deployer » pour installer les chemins de données qui supportent le VNET sur l'infrastructure ADN. Ce fonctionnement du « Request Handler » est illustré à la Figure 3 qui met également en avant une première phase dont l'objectif est de déterminer sur quel commutateur Openflow est rattachée chaque extrémité de lien virtuel. Ces informations sont obtenues des contrôleurs Openflow, qui exposent aux applications de contrôle du réseau ce type de service.

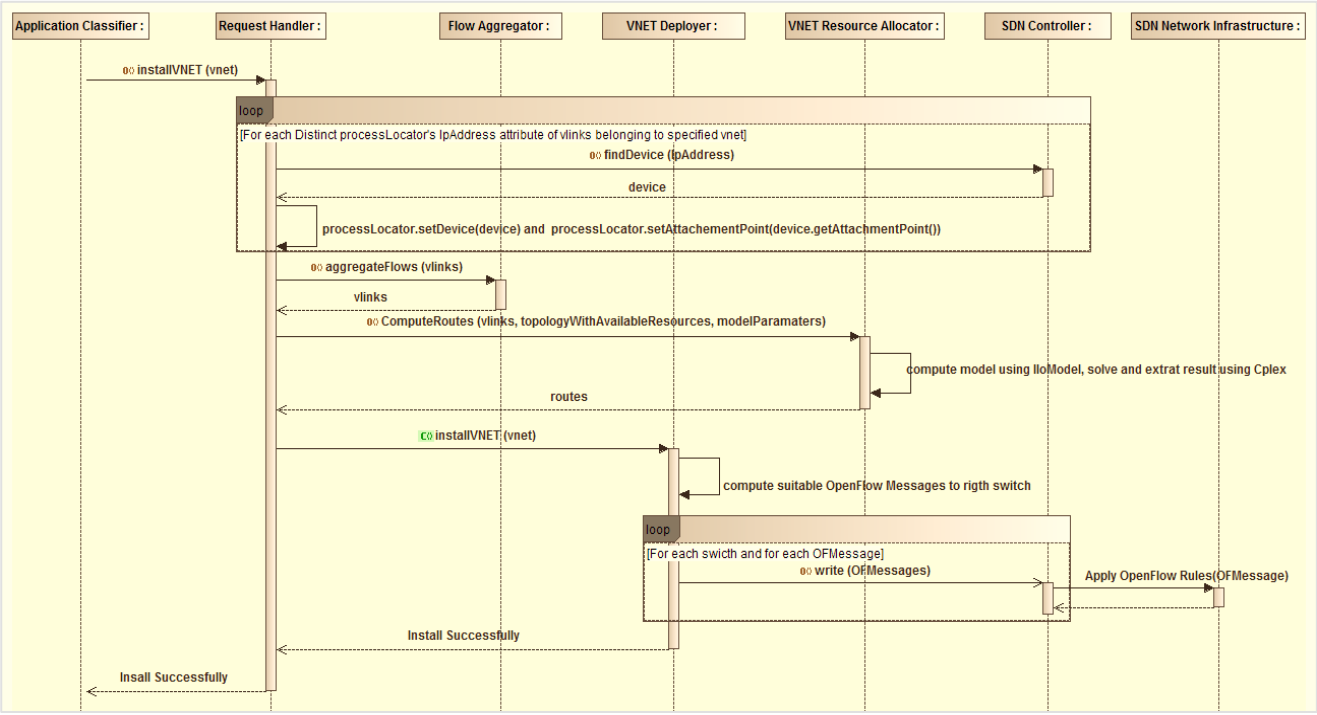


Figure 3 – Fonctionnement du « Request Handler » lors d'une demande d'établissement d'un VNET

2.2.2 FLOW AGGREGATOR

Par la possibilité offerte de pouvoir décrire finement les besoins applicatifs, l'approche ADN a comme avantage de pouvoir fournir les garanties de QoS souhaitées par les applications tout en utilisant efficacement les ressources. En revanche, il est clair que l'approche pose des problèmes d'échelle. Un aspect important de ce problème d'échelle est le nombre des entrées qui sont installées dans la table des flux pour supporter le service. En effet, étant basés sur la technologie TCAM, la taille des tables des flux des commutateurs Openflow actuels demeure limitée à quelques milliers d'entrées. Même si l'algorithme d'allocation prend en compte cette contrainte et cherche à minimiser le nombre d'entrées des tables des commutateurs SDN et à répartir leur charge (se référer à l'algorithme d'allocation de ressources décrit dans [1]), il serait conjointement intéressant d'agir en amont, c'est-à-dire lors de la construction du VNET à provisionner. C'est exactement le rôle du composant "*Flow Aggregator*". Il a pour tâche d'établir le VNET final à provisionner et pour lequel le « Resource Allocator » aura à déterminer les ressources à allouer, en

regroupant autant que possible les liens virtuels. Ces regroupements ont un coût en termes de consommation de ressources. C'est l'algorithme de ce composant (sur la base de la politique de gestion du réseau ADN) qui optimisera ce compromis entre ressources réseau gaspillées et nombre d'entrées installées dans les nœuds SDN.

2.2.3 VIRTUAL NETWORK RESOURCE ALLOCATOR

A la réception d'une demande d'établissement d'un VNET, ce composant a pour tâche de calculer l'ensemble optimal des chemins de données (avec les ressources réseau associées) à utiliser pour supporter les liens virtuels qui composent le VNET. Plusieurs critères d'optimisation et donc plusieurs algorithmes sont possibles. Parmi les critères considérés dans le livrable 1.2 figure la minimisation de la quantité de ressources allouées à chaque réseau virtuel ainsi que l'équilibrage de la charge entre les différents nœuds et liens de l'infrastructure physique. Avec ce critère, l'admissibilité des prochaines demandes d'allocation de ressources est favorisée. Pour les mêmes fins, le mécanisme de « path-splitting » peut être activé, à la demande pour certaines requêtes, lors du calcul des chemins de données. Cette possibilité permet qu'un lien virtuel soit supporté par plusieurs chemins physiques et donc agit pour augmenter l'admissibilité d'une requête d'établissement de VNET. Deux types de ressources réseau sont prises en compte dans le calcul : classiquement, la bande passante des liens mais également les ressources de commutation des nœuds qui sont représentées par la table des flux, de la table des groupes Openflow et la table des « meters » Openflow [12].
Enfin, il est à noter que ce composant gère également les demandes de mise à jour de VNET. Les demandes de retrait d'un VNET sont gérées entièrement par le composant "Request Handler".

2.2.4 VIRTUAL NETWORK DEPLOYER

L'objectif de ce composant est l'installation effective du VNET sur l'infrastructure SDN. Il prend en entrée les chemins de données calculées par le module d'allocation de ressources et génère les règles Openflow à appliquer au niveau de chaque commutateur Openflow en les soumettant au contrôleur via son interface nord puis en lui demandant de les installer sur les commutateurs. L'algorithme de ce composant a été décrit au livrable 1.2 [1].

2.2.5 APPLICATION CLASSIFIER

Comme indiqué ci-avant, le réseau ADN supporte aussi bien les applications qui expriment explicitement leur souhait d'utiliser un service ADN que celles qui ne le font pas. Ce composant a pour objectif d'identifier en temps-réel les applications qui ont le droit de prétendre aux services ADN, d'estimer leurs besoins actuels puis de soumettre la requête d'établissement ou de mise à jour de VNET correspondante. Les analyses de trafic réseau sont réalisées par des DPIs déployés à l'extrémité du réseau et qui sont sous le contrôle de l'« Application Classifier ».

2.2.6 AUTONOMIC MANAGER

L'objectif du composant « *Autonomic manager* » est d'ajouter certaines propriétés autonomiques à l'application de contrôle réseau "*ADN service provisioning*". Il gère les composants décrits ci-avant en implantant la boucle autonomique MAPE (Monitoring, Analysis, Planning and Execution) en se basant sur le cadre logiciel "frameself" développé au LAAS.
Sans être exhaustif, les scénarios préliminaires identifiés auxquels l' « *Autonomic Manager* » est programmé à réagir sont décrits ci-après :

- A la suite d'un changement de topologie réseau, il décide si une réallocation de ressources doit être enclenchée (propriété d'auto-réparation) ;
- En fonction des ressources réseau disponibles et de la requête de rajout ou de mise à jour d'un VNET, il paramètre l'algorithme d'allocation de ressources (par l'activation ou l'inhibition du « path-splitting ») et/ou de l'agrégation de flux ;
- Similairement, après des demandes de libérations de VNETs, il décide du moment où il est pertinent de re-calculer les ressources allouées aux VNETs présents pour mieux distribuer la charge des éléments réseau (auto-optimisation) ;
- Par ses actions de monitoring du trafic réseau, il détecte que le débit courant d'un lien virtuel a évolué et décide d'activer un processus d'estimation du trafic relatif à l'application (auto-configuration) pour ensuite soumettre une requête de mise à jour du VNET auquel le lien appartient.

2.3 Synthèse

Cette section a introduit l'architecture fonctionnelle de l'application de «ADN service provisionning » qui a été effectivement implémentée dans la tâche 2 du projet ADN. La Figure 4 résume la logique globale de ADN.

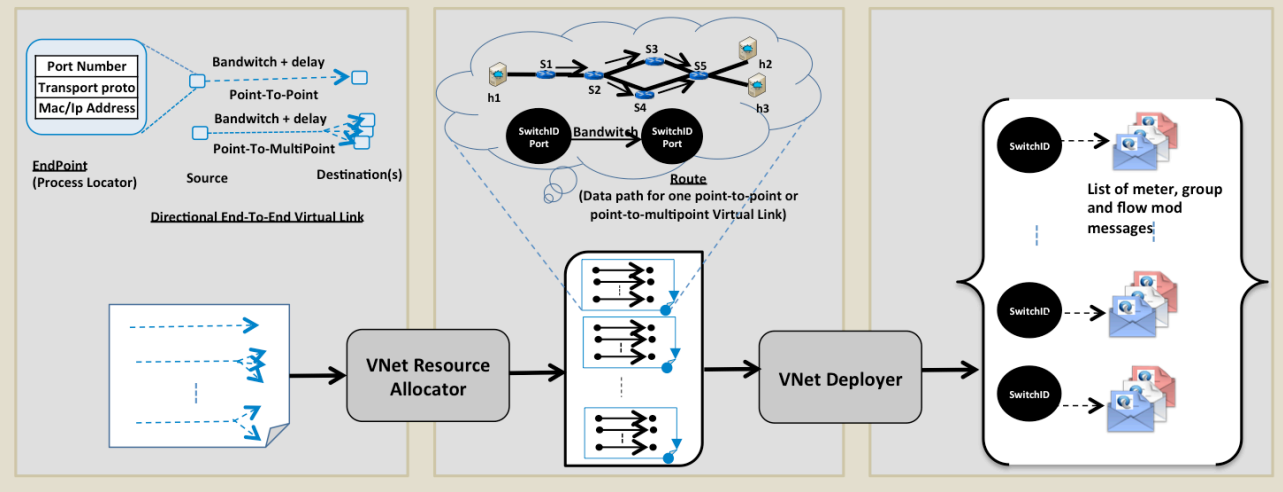


Figure 4 – Synthèse de l'approche ADN

Comme indiqué ci-avant, cette architecture se veut générale et non exclusivement dédiée aux applications DDS, même si les algorithmes de certains composants concernent ce type d'application. Avec cette architecture, d'autres instances de ces algorithmes peuvent ultérieurement être proposés et venir en complément de ceux déjà développés pour adresser d'autres applications.

3 PROTOTYPE ADN

L'objectif de cette section est de décrire les détails d'implémentation des composants logiciels mis en œuvre pour offrir aux applications DDS, les services réseaux de type ADN. Ces détails couvriront pour chaque composant, la description des services offerts et leurs dépendances requises, les messages échangés et la dynamique globale du système. En seconde partie, on détaillera la nature technique des interfaces exposant leurs services, leur structure interne et la description des principaux mécanismes associés à chaque composant.

3.1 Implémentation vue à l'échelle des composants et dynamique du système.

De l'architecture générale proposée et des exigences fonctionnelles associées à chaque composant, combinées aux choix techniques, découle l'implémentation du réseau ADN décrite en termes de composants dans le diagramme de la Figure 5 explicitant l'organisation et les dépendances d'exécution et de compilation entre les éléments du système.

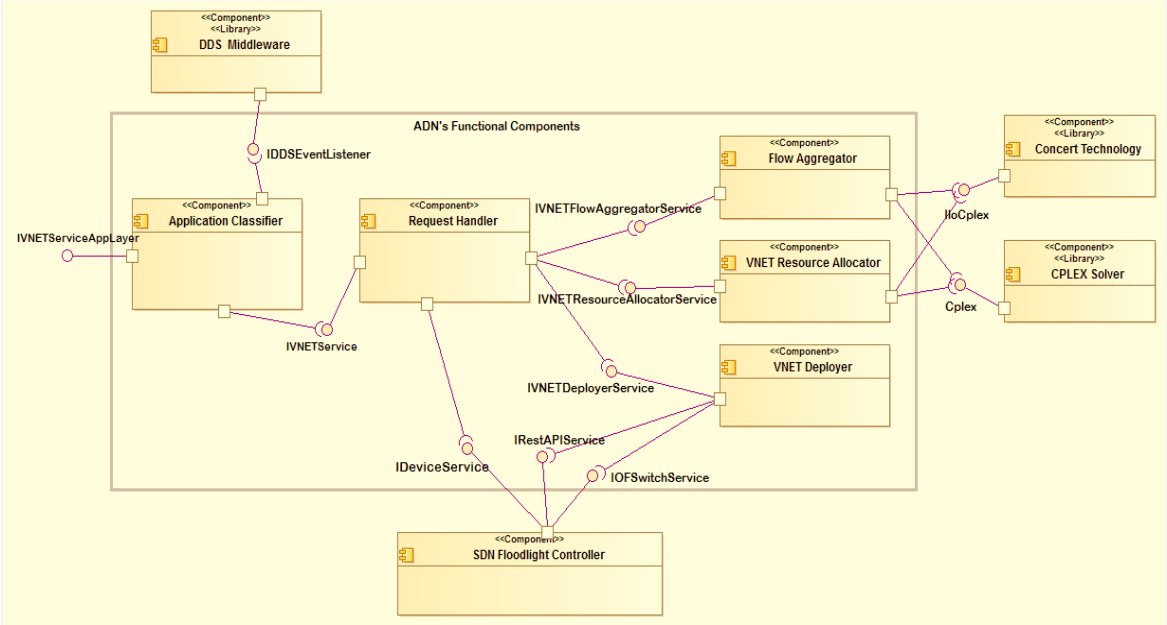


Figure 5 - Diagramme de composants du système ADN

Afin de percevoir la dynamique globale du système à ce niveau de granularité, les diagrammes de 0séquence présentés dans les Figure 7 et Figure 8, reprennent les signaux d'appel des opérations exposées par les interfaces (décrites à la Figure 6) des composants du système. Ils illustreront les interactions mises en œuvre pour satisfaire les scénarios de cas d'utilisation considérés dans le cadre du projet ADN. Ils considèreront l'utilisation (demande de création, de mis à jour ou de suppression de chemins de données (VNET)) du système par les applications de type DDS. Ces applications spécifient leurs besoins en utilisant une sémantique applicative de plus haut niveau que celle exposée par le modèle de service ADN (qui est de niveau réseau) et implique la mise en œuvre du composant *ApplicationClasssifier*.

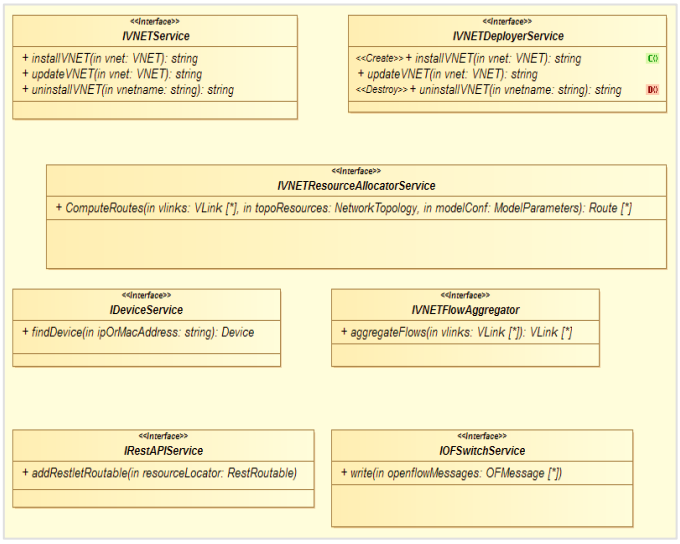


Figure 6 - Description fonctionnelle des interfaces

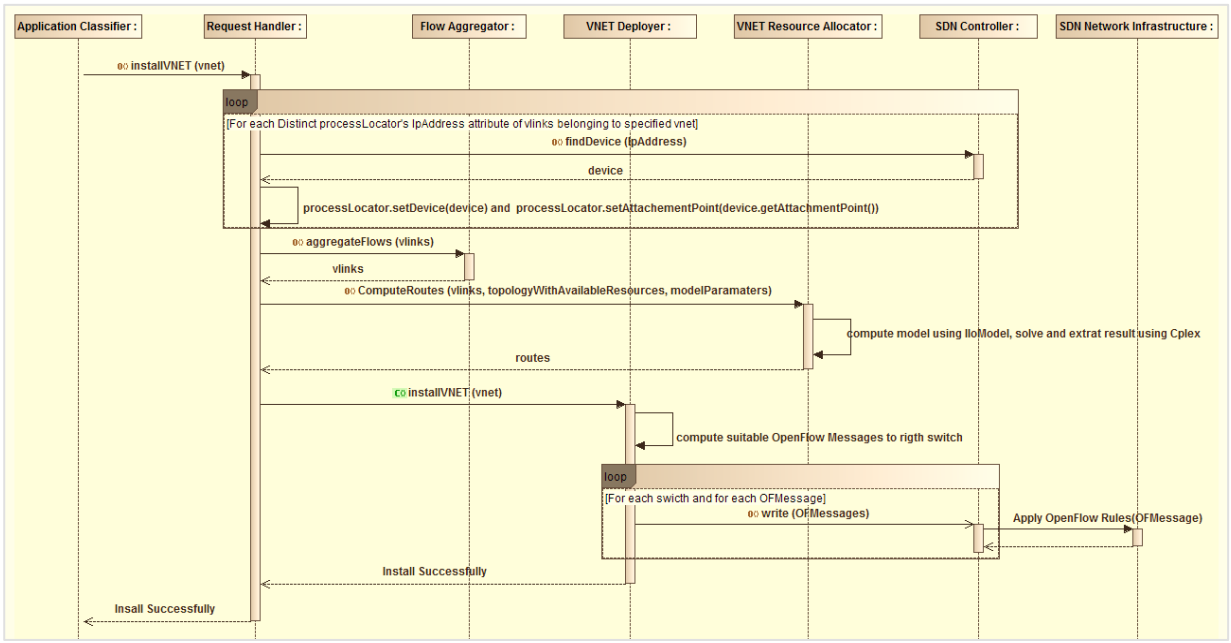


Figure 7 - Diagramme de séquence du cas d'utilisation « Install VNET »

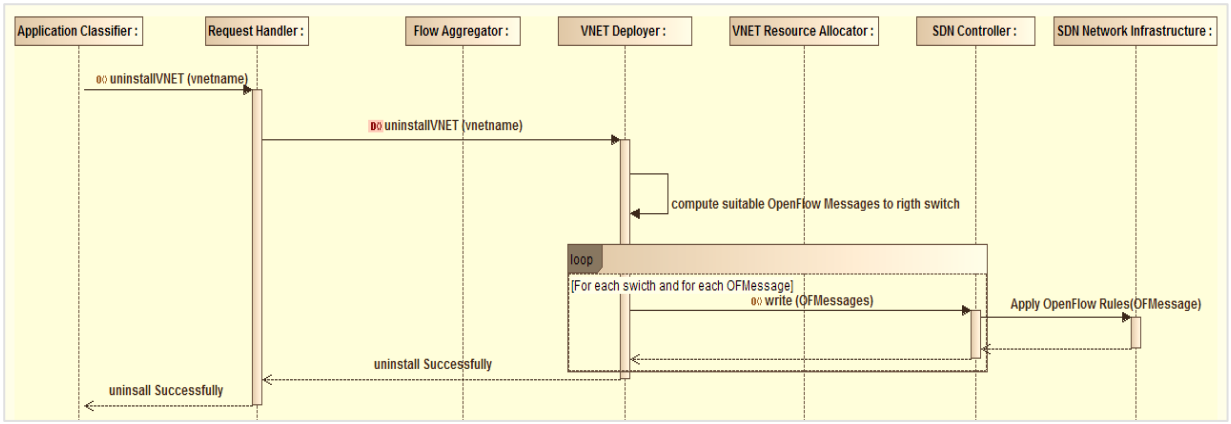


Figure 8 - Diagramme de séquence du cas d'utilisation « Uninstall VNET »

Les diagrammes de classes donnés dans les paragraphes suivants représentent les modèles de données sur lequel s'appuie le système ADN, en particulier la relation avec les composants externes. La Figure 9 représente l'interface de manipulation des réseaux virtuels, La Figure 11 représente les données liées au support de DDS, la Figure 17 concerne le support d'OpenFlow et la relation avec le contrôleur et la Figure 18 l'interface avec le solver CPLEX.
Ces derniers sont exploités à plusieurs étages du système, pour spécifier tour à tour les besoins de l'application en service réseaux de type ADN (**VNET** et ses **VLinks**), les chemins de données calculés (**Route** pour chaque **VLink**) par l'allocateur de ressource et les messages (**oFMessages**) de commandes OpenFlow générés par le *VNET Deployer* et à appliquer sur les commutateurs. Le diagramme de la Figure 9 présente le jeu de classes intégré qui a été considéré pour capturer ces données.

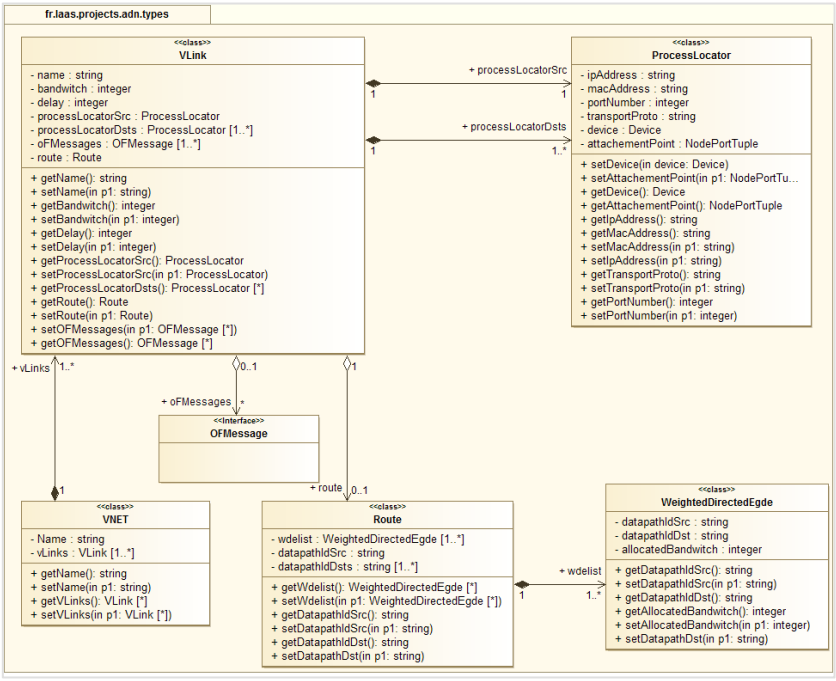


Figure 9 – Package de base et diagramme des classes du service ADN

3.2 Implémentation vue à l'échelle de chaque composant

La section précédente a présenté les grands composants du système, leurs dépendances et interfaces fonctionnelles. Elle a également évoqué son aspect dynamique en faisant référence aux diagrammes de séquences découlant des cas d'utilisation et matérialisant les interactions entre les composants et les tâches qu'ils exécutent pour réaliser ces cas d'utilisation. Cette section détaille la structure interne et les mécanismes des composants cités ci-après pour lesquels leurs implémentations sont actuellement abouties : *Application Classifier*, *VNET Deployer*, *VNET Resource Allocator*, *Request Handler*.

3.2.1 Application Classifier

Les responsabilités de ce composant ont été explicitées à la section 2.2.5. Dans cette rubrique, nous allons l'instancier pour les applications DDS. La Figure 10 présente sa structure interne.

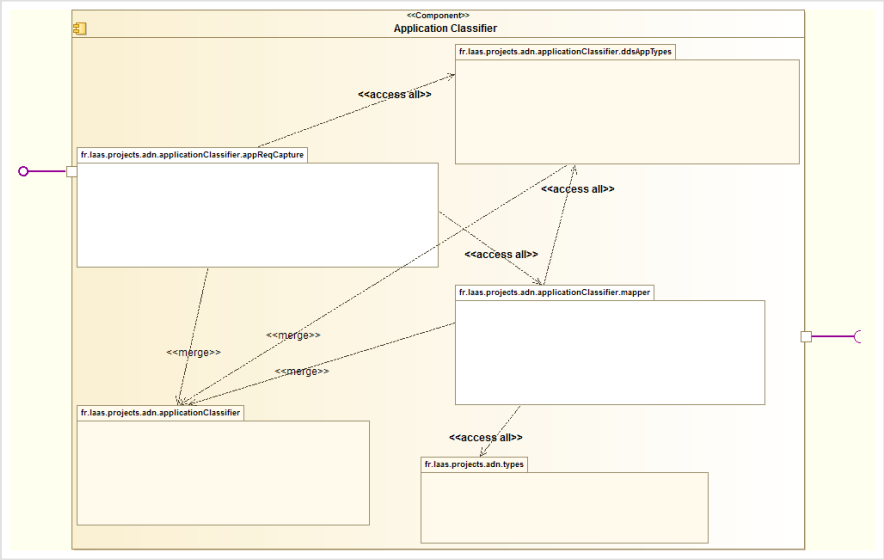


Figure 10 - Structure interne niveau package du composant Application Classifier

Le package « fr.laas.projects.adn.applicationClassifier.ddsAppTypes » contient les classes utilisées pour respecter les besoins en QoS niveau réseau des applications explicités en utilisant le formalisme DDS. Sa structure interne est présentée dans la Figure 11. Ce package est utilisé par deux autres packages : le package assurant la capture des besoins des applications l'exploite pour sauver les informations pertinentes capturées, puis consultera le second package qui l'utilise également, pour assurer leur traduction en service réseaux.

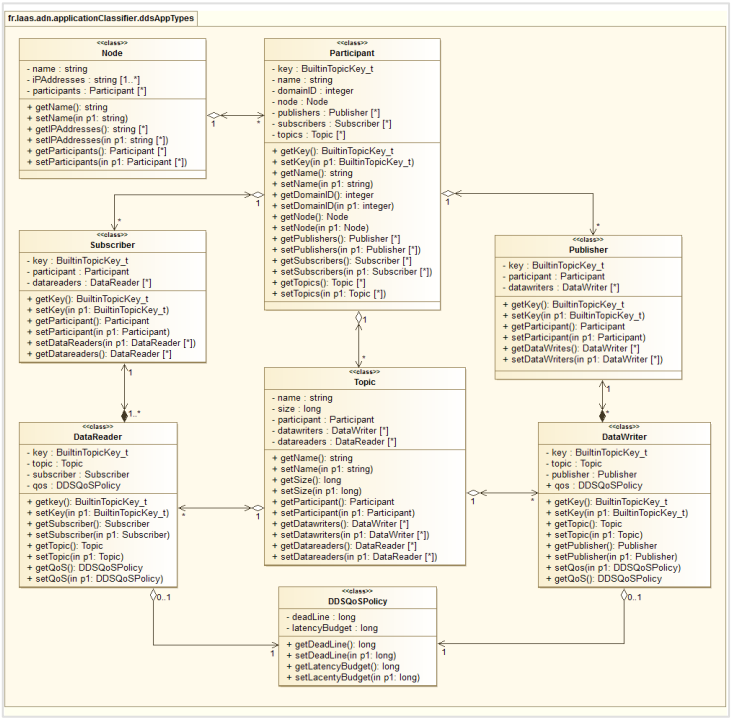


Figure 11 - Structure interne du package « fr.laas.projects.adn.applicationClassifier.ddsTypes »

Le package « fr.laas.projects.adn.applicationClassifier.appReqCapture » assure la capture des besoins des applications en exploitant l'API fournie par le middleware DDS, justifiant la dépendance observée dans le diagramme de composant de la Figure 5. La Figure 12 détaille sa décomposition interne. L'interface « DataReaderListener » est celle fournie par DDS et nécessite d'être implémentée pour capturer les besoins des applications. La classe abstraite « BuilinTopicDataListenerBase » implémente l'interface suscitée et implante le corps de la méthode

requise « on_data_available » en s'appuyant sur un mécanisme présenté à la section 3.2.1.1. Cette classe abstraite est étendue par le reste des classes de ce package qui réutilisent la méthode « on_data_avalaible ». Le package « fr.laas.projects.adn.applicationClassifier.mapper » est invoqué par le package assurant la capture des besoins des applications, pour les traduire et solliciter les services réseaux de type ADN auprès du composant *Request Handler* en utilisant les classes du package de base « fr.laas.projects.adn.types ». Le package « fr.laas.projects.adn.types » contient les classes de base du service ADN. La Figure 9 développe son contenu.

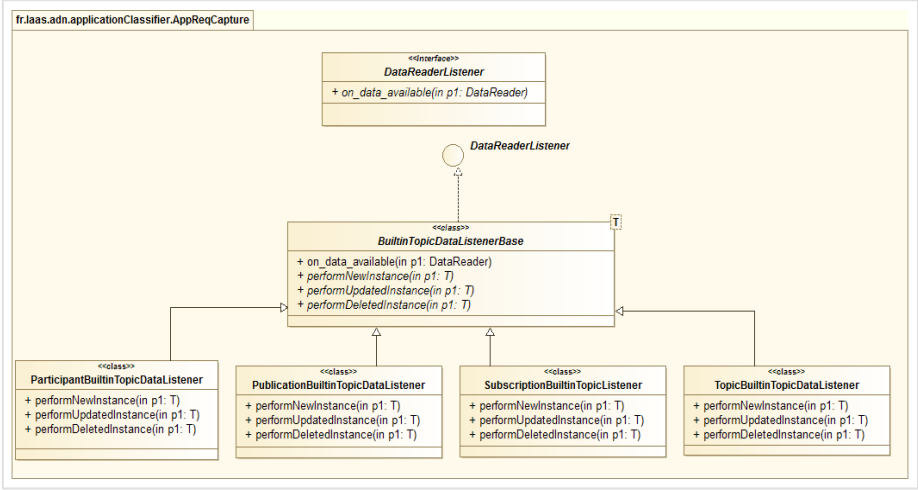


Figure 12 - Structure interne du package « fr.laas.projects.adn.applicationClassifier.AppReqCapture »

Le package « fr.laas.projects.adn.applicationClassifier.mapper » contient une seule classe conçue selon le pattern singleton et contient deux fonctions synchronisées : celle mettant en œuvre la traduction des besoins en service réseau et l'autre qui utilise l'interface du composant *Request Handler* pour solliciter ces services.

Dans la suite de cette section, nous allons présenter le mécanisme de capture des besoins des applications DDS et l'algorithme de retraduction en service réseau.

3.2.1.1 Capture des besoins de l'application

a) Rappel du problème

La couche Data Centric Publish Subscribe (DCPS) de DDS, fournit un support de QoS permettant aux applications de contrôler plusieurs aspects du service de distribution de leurs données afin de répondre à leurs besoins variés, et dynamiques. Ce contrôle se fait par le biais d'un nombre important de paramètres de Qualité de Service (désignés par DDS comme « QoS Policy »), ce qui confère à DDS un fort pouvoir d'expression. Ces paramètres spécifiés selon une approche orientée objet, bénéficient par conséquent des propriétés (réutilisable, extensible, …) liés à ce concept. Chaque entité DDS (DDS Entity) supporte toute ou une partie de ces paramètres. Les applications spécifient leurs exigences à l'échelle de chaque entité DDS, de façon déclarative via une API servant d'abstraction des mécanismes sous-adjacents mis en œuvre pour les garantir. Les exigences liées aux paramètres QoSDeadLine et LatencyBudget supportés par les entités DataWriter et DataReader et nécessitent une performance particulière de la part du réseau pour être garanties. En effet, le paramètre DeadLine indique à DDS la période maximale d'envoi ou de réception des échantillons d'un topic, selon qu'il soit configuré respectivement sur le DataWriter ou le(s) DataReader(s) associé(s) à ce topic. Ce qui impose que le réseau doit (1) disposer idéalement pour chaque topic des chemins de données de bout en bout avec suffisamment de bande passante et

respectant le délai maximal de transit. (2) être capable de maintenir ce chemin et le faire évoluer selon d'une part l'évolution de ses performances contrôlées par les paramètres DeadLine et LatenceBudget et d'autre part l'évolution de sa connectivité. L'arrivée spontanée de nouveaux consommateurs, le départ de ceux existant, la suppression du topic ou le départ volontaire peut faire évoluer cette connectivité.

b) Solution courante et limites

Actuellement, le middleware DDS responsable du service de distribution de données, compile ces besoins au niveau approprié de QoS que le réseau sous-jacent est capable de prendre en compte. Pratiquement, hormis les tags DSCP, ces paramètres ne sont pas pris en compte par les réseaux actuels. Les chemins de données sont généralement mis en œuvre en utilisant le multicast où de façon trivial un groupe représente une communication autour d'un topic et dont les paquets IP sont marqués au niveau du champ DSCP par l'identité du niveau service approprié. Ainsi, c'est en communiquant séparément avec les mécanismes de routage (supportant le multicast) et de QoS, que le middleware tente d'offrir le service promis.
Cette approche souffre d'un manque de flexibilité et offre des garanties souples.

c) Solutions proposées.

Approche 1

Cette approche préconise d'intégrer au niveau du middleware un composant pilote lui permettant d'avoir recours à une solution alternative lors de la compilation des besoins des applications, en utilisant le modèle de service réseaux implémenté par ADN.
Cette approche est intrusive et nécessite d'avoir accès au code source du middleware. Cette contrainte freine son adoption, du moins pour ce type d'application.

Approche 2 (celle adoptée)

Les implémentations actuelles du middleware DDS, n'étant pas conçues pour supporter les services offerts par les réseaux de nouvelles générations basés sur le paradigme SDN, et dans l'optique de surmonter des limites de l'approche précédente, nous avons choisi d'implémenter une solution transparente et interopérable. En référence au problème énoncé ci-avant, l'idée est de surveiller chaque application DDS, en accédant en temps réels aux topics disponibles. Pour chaque topic seront identifiés son DataWriter, ses DataReaders et leur objet QoS associé afin d'être notifié lorsque l'un des changements survient. Ce composant peut être perçu comme une méta-application qui a, de façon centralisée un regard sur toutes les entités DDS mis en œuvre dans chaque application ciblée par le réseau ADN. Pour atteindre cet objectif, nous n'aurons pas recours à un protocole ou mécanisme adhoc particulier, ou emprunté hors du périmètre du middleware DDS, afin de respecter l'exigence d'interopérabilité et de portabilité que nous cherchons à satisfaire. Par contre nous nous appuyons sur une fonctionnalité décrite dans la spécification DDS appelée Builtin Topics. Cette fonctionnalité associe à chaque entité DDS un topic spécial de type Builtin Topic qui est géré implicitement par le middleware et via lequel il publie les informations concernant toutes les instances des entités (Topic, DataWriter, DataReader, Participant) créées par les applications DDS s'exécutant localement. Ces informations sont mises à disposition de toute application (distante ou locale) intéressée et sont accessibles en utilisant les mêmes mécanismes d'accès aux données tels que les Listeners, ce qui évite d'introduire une nouvelle API spécifique. Il ne reste plus qu'à créer des DataReaders qui s'abonnent à chacun de ces builtin topics et capturer ainsi en temps réels les besoins des applications. Comme tout topic, chaque builtin topic a un nom et une structure de données typée établie à l'image de l'entité qu'il modélise. On distingue le builtin Topic de nom DCPSTopic modélisant l'entité Topic, un autre de nom DCPSPublication

modélisant l'entité DataWriter, puis celui de nom DCPSSubscription modélisant l'entité DataReader et enfin le builtin topic DSCPParticipant modélisant l'entité Participant.

Le

Tableau 1 détaille leur contenu. Les données d'information portant sur un DataWriter ou un DataReader référencient le topic auquel il est associé et la QoS spécifiée en terme de DeadLine et LatenceBudget. Cependant les informations sur leur adresse IP, le protocole de transport et le numéro de port n'y sont pas. L'implémentation RTI Connext du middleware DDS les fournit comme extension. Pour cette raison, elle a été adoptée dans nos développements.

A cette étape nous avons toutes les informations pour capturer les besoins de l'application. Toutefois, il faut prendre en compte la dynamique. Toute instance de chacune de ces entités est identifiée par une clé et chaque modification apportée sur l'instance (par exemple le changement de son paramètre QoS DeadLine ou sa suppression), se traduit par la publication d'un échantillon. La Figure 13 présente à titre illustratif, la relation entre un *builtin topic*, ses instances et les échantillons émis en cas de changement. Il est donc nécessaire d'analyser ces échantillons pour déduire la création d'une nouvelle instance, sa suppression et sa modification.

BuiltinTopic name : **DCPSTopic**	
Field name	Type
key	BuiltinTopicKey_t
name	string
type_name	string
durability	DurabilityQosPolicy
durability_service	DurabilityServiceQosPolicy
deadline	DeadlineQosPolicy
latency_budget	LatencyBudgetQosPolicy
liveliness	LivelinessQosPolicy
reliability	ReliabilityQosPolicy
transport_priority	TransportPriorityQosPolicy
lifespan	LifespanQosPolicy
destination_order	DestinationOrderQosPolicy
history	HistoryQosPolicy
resource_limits	ResourceLimitsQosPolicy
ownership	OwnershipQosPolicy
topic_data	TopicDataQosPolicy

BuiltinTopic name : **DCPSPublication**	
Field name	Type
key	BuiltinTopicKey_t
participant_key	BuiltinTopicKey_t
topic_name	string
type_name	string
durability	DurabilityQosPolicy
durability_service	DurabilityServiceQosPolicy
deadline	DeadlineQosPolicy
latency_budget	LatencyBudgetQosPolicy
liveliness	LivelinessQosPolicy
reliability	ReliabilityQosPolicy
ownership	OwnershipQoSPolicy
destination_order	DestinationOrderQosPolicy
user_data	UserDataQoSPolicy
time_based_filter	TimeBasedFilterQoSPolicy
presentation	PresntationQoSPolicy
partition	PartitionQoSPolicy
topic_data	TopicDataQoSPolicy
group_data	GroupDataPolicy

BuiltinTopic name : **DCPSSubscription**	
Field name	Type
key	BuiltinTopicKey_t
participant_key	BuiltinTopicKey_t
topic_name	string
type_name	string
durability	DurabilityQosPolicy
durability_service	DurabilityServiceQosPolicy
deadline	DeadlineQosPolicy
latency_budget	LatencyBudgetQosPolicy
liveliness	LivelinessQosPolicy
reliability	ReliabilityQosPolicy
transport_priority	TransportPriorityQosPolicy
lifespan	LifespanQosPolicy
destination_order	DestinationOrderQosPolicy
presentation	PresntationQoSPolicy
history	HistoryQosPolicy
partition	PartitionQoSPolicy
topic_data	TopicDataQoSPolicy
group_data	GroupDataPolicy

Tableau 1 - Structure de données associées aux Builtin Topics

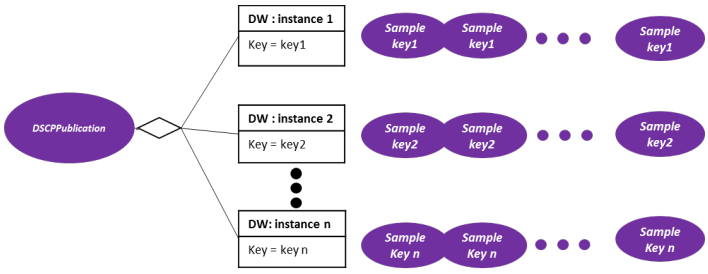


Figure 13 - Relation entre un builtin topic, ses instances et leurs échantillons

La spécification DDS définit une structure de données *SampleInfo* apportant les informations sur l'échantillon reçu permettant de savoir s'il provient d'une nouvelle instance ou s'il s'agit d'une modification ou d'une suppression d'une instance existante. Nous avons utilisé cette fonctionnalité pour capturer et maintenir les besoins des applications. La Figure 14 présente le diagramme d'état permettant d'analyser l'échantillon d'une instance qui a été reçue et de déduire si c'est une nouvelle instance (donc une nouvelle entité DDS), si c'est une modification d'une instance existante (cas de changement de la QoS d'une entité existante) ou sa suppression (cas du départ d'une entité).

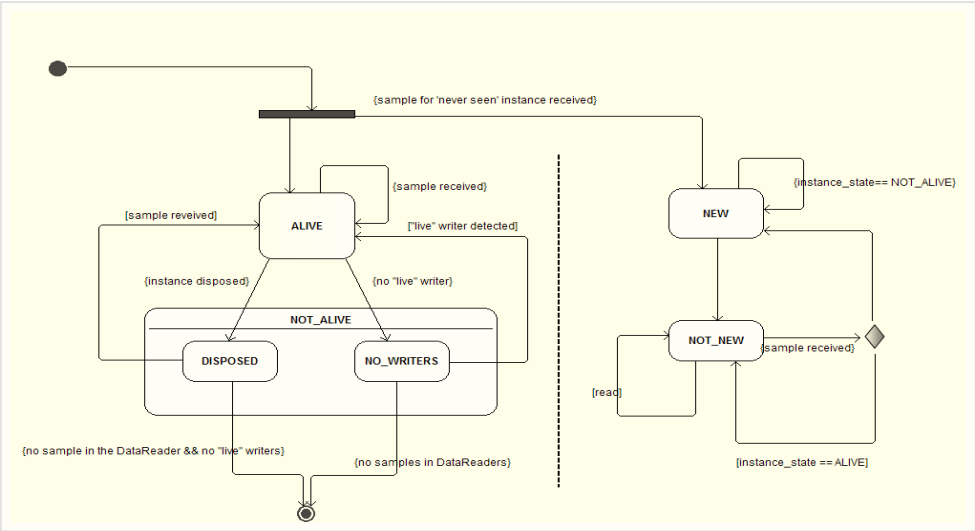


Figure 14 - diagramme d'état d'une instance et de sa vue

Ce mécanisme est au cœur de l'implantation de la fonction template « on_data_available ». Il permet de suivre l'évolution des besoins des applications. Par ailleurs, il nécessite que le composant exécute le middleware DDS en tant que participant observateur du domaine DDS pour pouvoir consommer ses services.

3.2.1.1 Traduction des besoins de l'application en service réseau

L'algorithme de traduction des besoins de l'application est implémenté dans l'unique classe du package « fr.laas.projects.adn.applicationClassifier.mapper ». Les détails de cet algorithme ont été explicités dans le livrable 1.2.

3.2.2 VNET DEPLOYER

Ce composant s'appuie sur la technologie REST (REpresentational State Transfert) pour exposer ses services (ceux-ci sont décrits dans la Figure 6). Le

URL	Description	Méthode HTTP	Arguments
http://controlerAddress:8080/wm/vnetdeployer/install-vnet/json	Crée le VNET	POST	Description du **VNET** au format json. dans le corps de la requête.
http://controlerAdress:8080/wm/vnetdeployer/uninstall-vnet/<vnetname>/json	Supprime le VNET	DELETE	**vnetname** : identifie de façon unique chaque VNET installé.
http://controlerAddress:8080/wm/vnetdeployer/update-vnet/json	Met à jour le VNET	PUT	Description du VNET au format json dans le corps de la requête.

Tableau 2 reprend la définition de ces services au sens REST.

http://controlerAddress:8080/wm/vnetdeployer/update-vnet/json	Met à jour le VNET	PUT	Description du VNET au format json dans le corps de la requête.

Tableau 2 - Description des services exposés via REST

Le contrôleur choisi pour envoyer les messages de commandes OpenFlow aux commutateurs de l'infrastructure est *Floodlight*. La particularité de ce contrôleur réside sur son support des fonctionnalités OpenFlow requises pour instancier les chemins de données. Avec sa compatibilité complète avec la version 1.3 d'OpenFlow et des bundles introduits dans la version 1.4, en plus de son interopérabilité avec le Switch logiciel OFSoftSwitch13 choisi, il supporte les *meters* et les groupes de type "*all*" et "*select*" indispensables pour implémenter le multicast et le multipath utilisés dans nos algorithmes. De plus, il offre un nombre important de services réseaux de base maintenus par une communauté active intégrant les acteurs du monde industriel et académique. Parmi ses services offerts, on décompte un service permettant aux fonctions de contrôle d'exposer leurs services propres via la technologie REST, moyennant un cout de développement considérablement réduit. En revanche, la fonction de contrôle réseau doit s'interfacer avec son API JAVA native et donc sera couplée à la plateforme Java. L'un des avantages de cette approche est de s'affranchir de l'overhead induit par une approche qui donne la priorité au découplage et un autre est la possibilité de tirer parti de la modularité du contrôleur *Floodlight*. C'est le choix qui a été effectué pour le développement du *VNET Deployer* et qui justifie la dépendance de ce composant au contrôleur SDN observée dans le diagramme de composant de la Figure 5. Dans cette dépendance il consomme le service *IOSwitchService* du contrôleur réseau pour la livraison des messages OpenFlow de configuration et utilise un service spécial appelé *RESTAPIService*, lui permettant d'exposer le plus simplement possible ses propres services.
Le *VNET Deployer* est un module au sens *Floodlight* et la Figure 15 montre son positionnement dans l'architecture de ce contrôleur. Il est donc fourni avec le contrôleur au format JAR (Java ARchive).

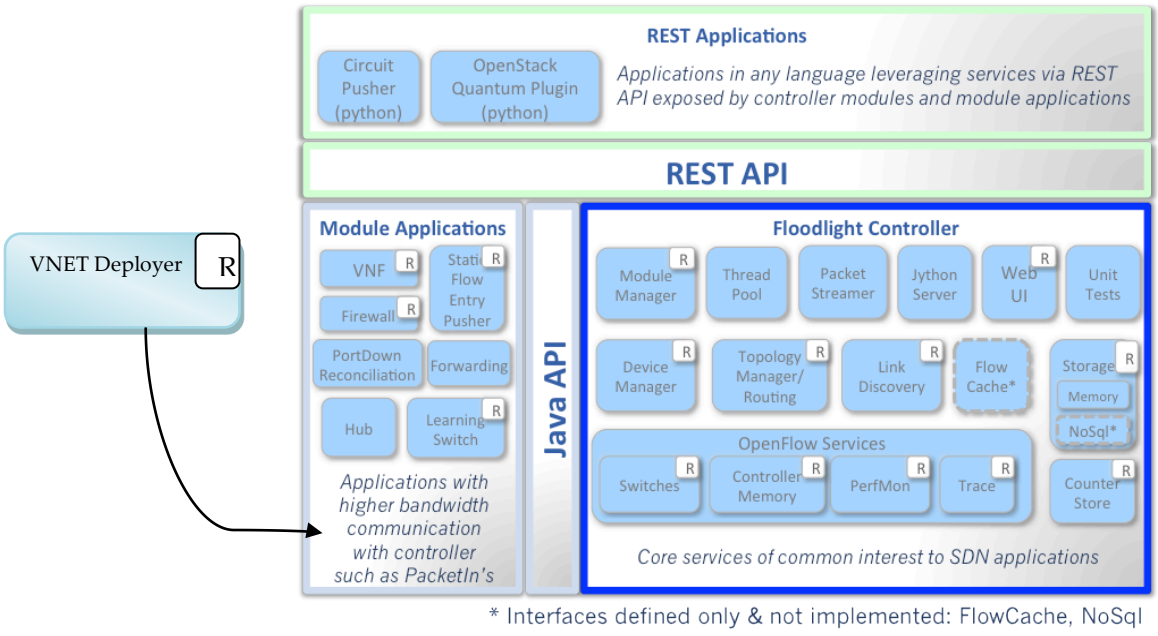


Figure 15 - Architecture du contrôleur réseau Floodlight

Floodlight utilise le framework Restlet comme implémentation REST. Ce framework est reconnu comme étant très léger car ne nécessite pas un serveur applicatif ou web particulier pour s'exécuter. C'est un environnement à part entière capable d'être déployé aussi bien de façon autonome que sur des environnements plus classiques. En tant que framework de la plateforme

java, il requiert à minima l'environnement java standard et une JVM (Java Virtual Machine) pour s'exécuter. Son principe est le suivant :

Chaque requête entrante est :

- acceptée par un composant central (ce composant est pris en charge par *Floodlight*) et qui hérite de la classe "Application" fournie par le framework.
- routée selon des règles de correspondance spécifiées par une classe propre au composant *VNET Deployer* et qui est une spécialisation de la classe "Router" également fournie par le framework.
- puis traitée par une unité d'exécution généralement référencée par le terme *Ressource* ayant pour classe mère la classe "Restlet" et chargée de traiter la requête en exécutant dynamiquement les traitements associés à la requête. Enfin le résultat est retourné au format spécifié par la requête au format JSON. Cette classe Ressource est prise en compte par le *VNET Deployer*

La Figure 16 présente la décomposition en package de ce composant et la répartition de ces classes dans leur package respectif. Le package « fr.laas.projects.adn.types » est celui de base contenant les classes du service ADN. Sa structure interne est présentée à la Figure 9. Le package « fr.laas.projects.adn.vnetdeployer.web » contient les classes précédemment évoquées à l'énoncé du principe de fonctionnement du framework *Restlet.* Ce package contient les classes permettant d'exposer les services via REST. Le package « fr.laas.projects.adn.vnetdeployer.webserializer » comme son nom l'indique, contient les classes permettant de sérialiser les objets instances des classes du package « fr.laas.projects.adn.types » au format JSON. Enfin le package « fr.laas.projects.adn.vnetdeployer » contient les classes permettant à ce composant de s'interfacer avec le contrôleur *Floodlight* et celles implémentant les algorithmes permettant le traitement proprement dit de la requête. Ces algorithmes sont contenus dans la version précédente du livrable. La Figure 17 présente le diagramme de ses classes principales.

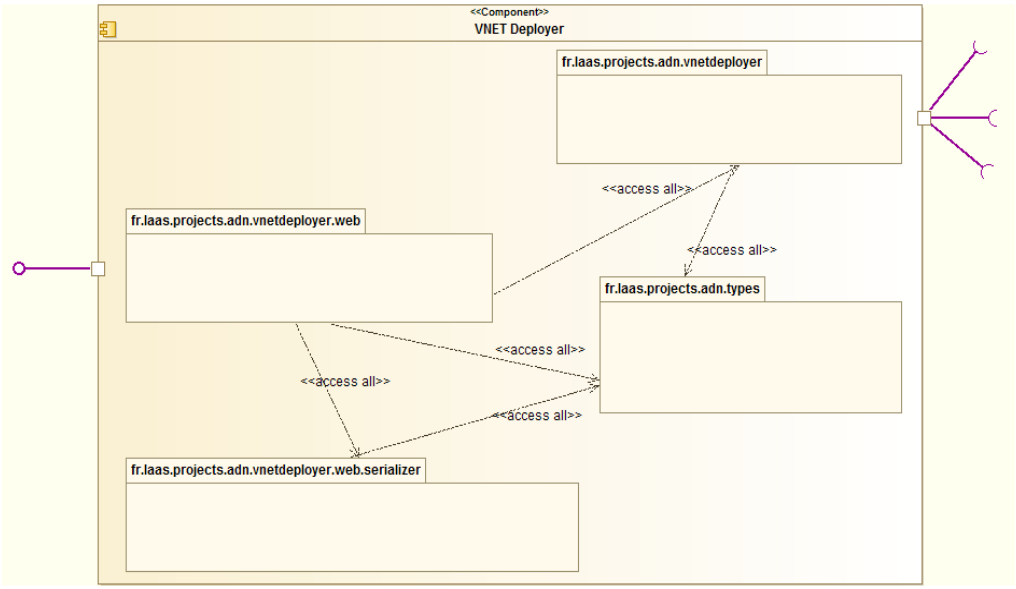


Figure 16 - Structure interne niveau package du composant VNET Deployer

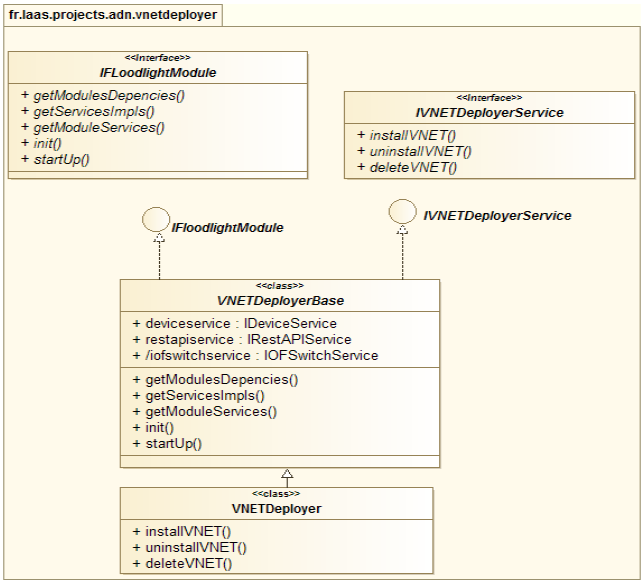


Figure 17 - Structure interne du package « fr.laas.projects.adn.vnetdeployer »

3.2.3 VNET RESOURCE ALLOCATOR

Ce composant est implémenté en C++ et livré comme un exécutable. Par conséquent, les services qu'il offre son consommé via des API permettant la communication inter processus. Par exemple en java cette API est mise en œuvre par le package « java.lang.runtime ». Pour offrir son service de calcul de chemin donné, il s'appuie sur deux étapes : la première prend en charge la modélisation de l'algorithme conformément à la couche de modélisation de CPLEX nommée *Concert Technology* et la seconde s'occupe de résoudre le problème d'optimisation modélisé repose sur le solveur *CPLEX*. Ce qui justifie sa dépendance à ces composants. Son algorithme a été développé dans le livrable 1.2. Ce composant prend comme paramètre : la spécification des liens virtuels pour lesquels les chemins de données doivent être calculés, la topologie réseau et les ressources disponibles passées via un fichier texte au format GML, les paramètres de l'algorithme et ceux du modèle. Il retourne pour chaque lien virtuel, les chemins de données calculés dans un objet de la classe *Route*. La Figure 18 décrit le diagramme des classes utilisées pour la modélisation et la résolution du modèle.

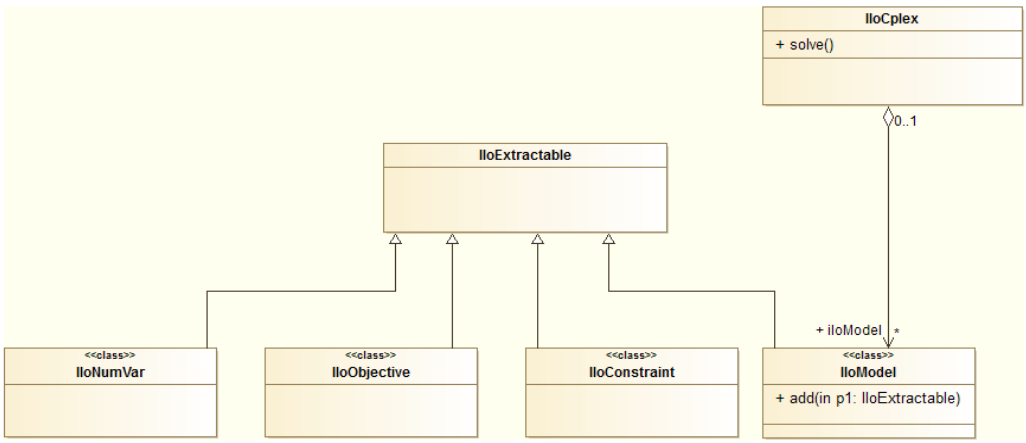


Figure 18 - Diagramme des classes de Concert technology et de CPLEX

3.2.4 REQUEST HANDLER

Ce composant est implémenté en java et livré dans un jar commun avec le composant *Application Classifier* qui le sollicite. De par son rôle d'orchestrateur des requêtes, il implémente les clients nécessaires pour consommer les services des autres composants. Sa décomposition en package est détaillée dans la Figure 19.

Le package « fr.laas.projects.adn.requesthandler » contient les classes permettant de réceptionner les requêtes et de les analyser et d'invoquer dans le bon ordre les services des autres composants. Les diagrammes de séquences des Figure 7 et Figure 8 montrent les exemples de schémas d'orchestration que ce composant réalise en fonction du type de requête. Pour invoquer les services du *VNET Resource Allocator* (via une interface de communication locale inter processus) et du *VNET Deployer* (via le REST) il utilise respectivement les programmes clients contenus dans les packages « fr.laas.projects.adn.requesthandler.vnetresourceallocatorclient » et « fr.laas.projects.adn.requesthandler.vnetdeployerclient ».

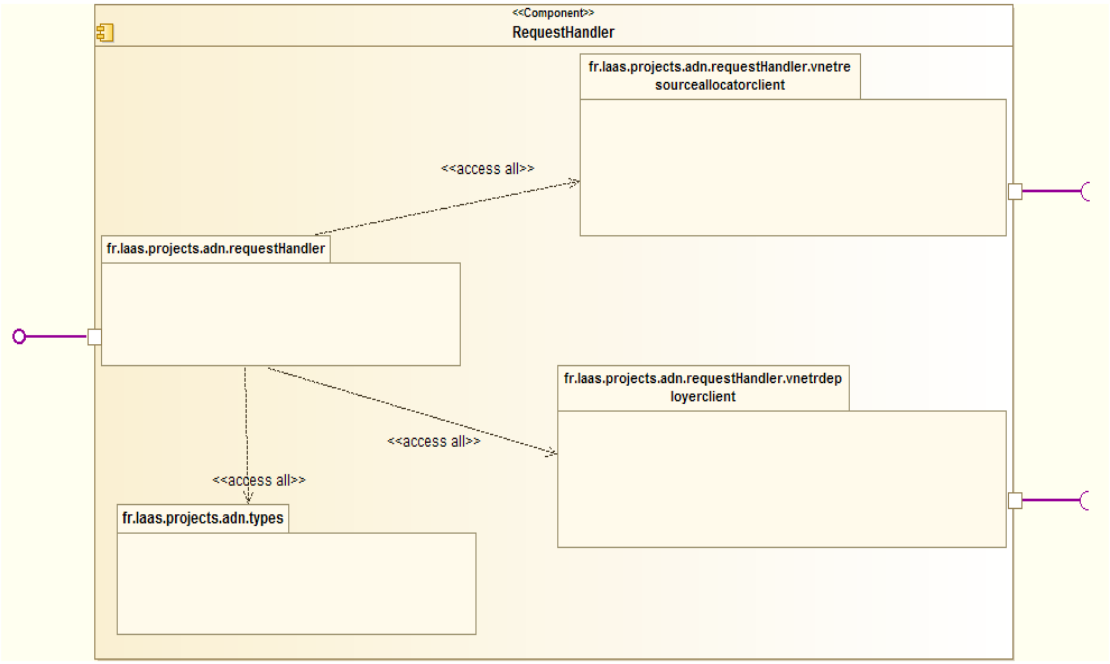


Figure 19 - Structure interne niveau package du composant Request Handler

4 DEMONSTRATEUR ADN

4.1 Introduction

Nous rappelons que l'application civile considérée pour le projet ADN est une application de simulation distribuée impliquant plusieurs simulateurs de véhicule dirigés par des apprenants qui participent à un exercice ou une formation à la conduite en groupe inspirée d'une des solutions commercialisées par ECA FAROS [16] (voir Figure 20). Des postes instructeurs recueillent les données de certains simulateurs et les renvoient à l'écran pour le compte d'instructeurs en charge de superviser la formation. Enfin, un serveur de simulation récupère et archive l'ensemble des données des simulateurs pour un éventuel rejeu de la simulation. Les instructeurs ont la possibilité d'établir des communications audio avec les apprenants. Ils ont également la possibilité d'activer l'envoi d'un flux vidéo depuis une caméra localisée sur le poste simulateur de véhicule et qui filme l'apprenant vers le poste instructeur.
L'application considérée repose sur l'intergiciel DDS (Data Distribution Service) de l'OMG (Object Management Group) pour distribuer les flux de données applicatives relatifs à la simulation. Les échanges audio et vidéo ne reposent pas sur DDS. Pour cette raison, nous nous focaliserons dans la suite sur les échanges de données en lien avec la simulation. Etant donnée, l'impossibilité de redévelopper nos propositions à partir des simulateurs ECA Faros, le prototype s'appuiera sur des pseudo-simulateurs (ou des émulateurs) qui chercheront à reproduire les données réelles envoyés par ces simulateurs sur le réseau.

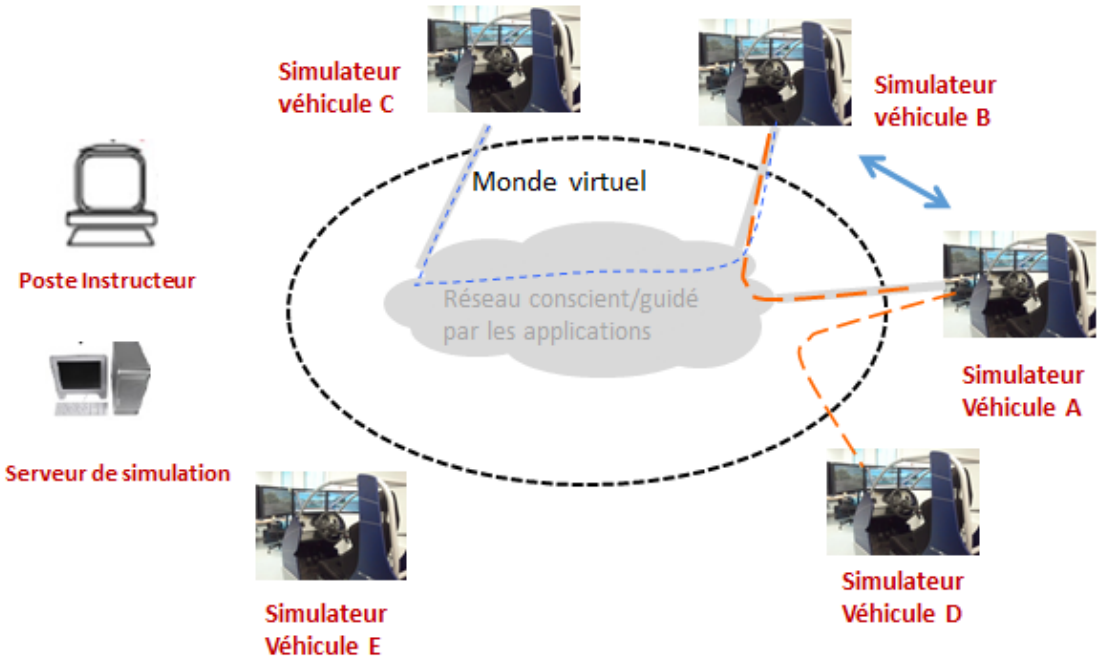


Figure 20 – Application de simulation considérée

En commun accord avec le partenaire DGA, le démonstrateur :

- Implique 5 pseudo-simulateurs de véhicule, un poste instructeur et un serveur de simulation ;
- Reposera sur les données issues des simulateurs de véhicule commercialisés par ECA Faros. Ces données sont disponibles au LAAS et concernent certains simulateurs commercialisés en 2011 ;
- Le réseau support considéré sera un réseau de campus équivalent à celui du LAAS/CNRS ;

- Le scénario de simulation qui sera considéré suppose un modèle de mobilité des simulateurs plutôt déterministe basé sur un plan routier (autour du LAAS) dans lequel les pseudo-simulateurs évolueront.
- La définition des besoins instantanés en QoS relatifs à la distribution des données des pseudo-simulateurs reposera sur la comparaison de la distance entre pseudo-simulateurs à des seuils préétablis. Le choix de ces seuils ne sera pas motivé (faute d'expertise métier sur ces aspects).
- Le paramètre vitesse de certains pseudo-simulateurs pourra être varié afin de considérer plusieurs scénarios applicatifs.

Le plan de cette section se présente comme suit. Nous commencerons par décrire l'application DDS considérée en explicitant ses flux de données (via la définition des topics DDS) et leurs besoins en QoS. Nous décrirons ensuite l'infrastructure réseau considérée et sa mise en œuvre dans le démonstrateur. Enfin, nous préciserons le scénario applicatif considéré par le démonstrateur, suivi, des résultats ou observations qui illustrent le fonctionnement du réseau ADN.

4.2 Application dynamique DDS considérée

Les données échangées entre les différents acteurs de l'application (simulateurs de véhicule, poste instructeur et serveur de simulation) reprennent les données des simulateurs de véhicules produits par ECA Faros. Elles sont principalement composées de trois types : des données de simulation, des données pédagogiques et données de contrôle. Elles sont organisées en 16 Topics DDS décrits ci-après.

4.2.1 DESCRIPTION DES TOPICS

La Figure 21 recense le nom des différents topics DDS de l'application en les organisant selon leur profil : périodiques, apériodiques, ou ponctuels et éphémères car échangés à l'initialisation de l'application (voir Figure 21).

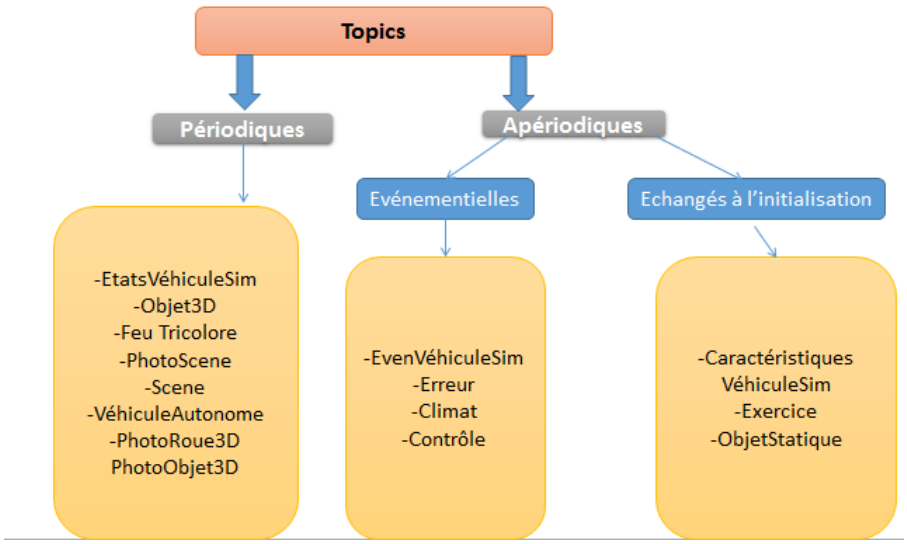


Figure 21 – Topics DDS relatifs à l'application considérée

La définition de ces topics est renvoyée en annexe, suit, seulement une explication intuitive de la signification de chaque topic :

- **Position :** C'est un topic qui montre la position de chaque simulateur au cours de la simulation. Ce topic sera échangé entre les simulateurs afin que chaque simulateur connaisse la position relative des autres simulateurs.
- **Objets3D :** C'est un topic générique qui présente tous les objets appartenant à l'environnement simulé à savoir les véhicules simulés et autonomes, les objets statiques, les feux tricolores et la scène simulée où se déroule l'exercice d'apprentissage. Il contient les données d'état en trois dimensions, elles indiquent alors principalement les caractéristiques 3D de chaque type d'objet à savoir son orientation, son rayon de visibilité et autres. Ce topic est produit par le serveur de simulation et consommé par les simulateurs.
- **CaractéristiquesVéhiculeSim :** Ce topic permet de transmettre une fois à l'initialisation de l'exercice les informations relatives à tous les véhicules simulés impliqués dans l'exercice (exemple le type moto ou auto, la masse,...). Ces données sont statiques et ne subissent aucune mise à jour durant toute la durée de vie de l'exercice. Ce topic est produit par les simulateurs pour le compte du serveur de simulation.
- **EtatsVéhiculeSim :** Ce topic contient les données d'état des véhicules simulés (sa position, sa vitesse) participant à l'exercice de simulation en question. Ces données sont échangées régulièrement afin d'avoir un état global du système. Ce topic est produit par chaque simulateur pour d'autres simulateurs, le serveur de simulation et le poste instructeur.
- **FeuTricolore :** Ce topic contient les données d'état de chaque feu tricolore (exemple sa couleur) existant dans l'environnement simulé. Ce topic est transmis par le serveur de simulation aux simulateurs et au poste instructeur.
- **Scene :** Ce topic contient les données de contrôle de la scène simulée où se déroule l'exercice d'apprentissage. elles sont échangées suite à un changement survenu. Ce topic est produit par le poste instructeur et consommé par les simulateurs et le serveur de simulation.
- **VéhiculeAutonome :** Ce topic contient toutes les données relatives à chaque véhicule autonome appartenant à l'exercice. Il est transmis du serveur de simulation vers chaque simulateur.
- **EvenVéhiculeSim :** Ce topic est échangé suite à un évènement sur un véhicule simulé appartenant à l'exercice (exemple panne). Ce topic est produit par les simulateurs et consommé par les autres simulateurs, le serveur de simulation et le poste instructeur.
- **ObjetStatique :** Ce topic contient les données caractérisant chaque objet statique (exemple panneau de signalisation) appartenant à l'exercice. Ce topic est produit par le serveur de simulation pour le compte des simulateurs et du poste instructeur.
- **Erreur :** Ce topic contient les données pédagogiques. Il est échangé suite à la détection d'une erreur individuelle ou collective commise par un ou plusieurs apprenants. Il recense le type d'erreur, sa criticité, et les apprenants concernés. Ce topic est transmis du serveur de simulation vers le poste instructeur concerné.
- **Exercice :** Ce topic contient les données décrivant les paramètres (par exemple le lieu, l'heure) de l'exercice. Elles sont mises à jour par un poste instructeurs. Il est transmis d'un poste instructeur vers le serveur de simulation et vers l'ensemble des simulateurs.
- **Climat :** Ce topic contient les données relatives au climat (par exemple intensité des précipitations) de l'exercice de conduite. Il est mis à jour suite à une décision prise par un poste instructeur pour changer le contexte de l'exercice d'apprentissage. Il est transmis d'un poste instructeur vers le serveur de simulation et vers l'ensemble des simulateurs.

- **Contrôle :** Ce topic permet de notifier un changement survenu à l'état global du système (exemple ajout d'un obstacle à la scène). Il est transmis d'un poste instructeur vers le serveur de simulation et les simulateurs.
- **PhotoScene :** Ce topic contient les données d'état de la scène simulée où se déroule l'exercice. Ces données sont échangées régulièrement pour maintenir une vue globale de la scène simulée et surtout pour garantir la synchronisation et la cohérence entre les différents participants. Ce topic est transmis du serveur de simulation vers les simulateurs.
- **PhotoRoue3D :** Ce topic permet d'avoir régulièrement les données 3D d'état de chaque roue de chaque véhicule simulé. Ce topic est échangé entre les simulateurs.
- **PhotoObjet3D :** De même pour ce topic, il contient les données 3D d'état de chaque Objet appartenant à l'exercice. Ces données sont échangées régulièrement. Ce topic est transmis du serveur de simulation aux simulateurs.

4.2.2 SPÉCIFICATION DES QOS DES TOPICS

DDS offre aux applications la possibilité de décrire finement, à travers plusieurs paramètres de QdS (appelés « QoS policies » dans DDS), le service DDS attendu [15]. Les paramètres de QdS que nous considérons sont: ""History", "Durability", Reliability", "Latency Budget" et "Deadline".

Pour l'ensemble des topics de l'application, les paramètres « History » et « Durability » sont respectivement fixés à « KEEP_LAST » et « VOLATILE ». En effet, se limiter au niveau du middleware DDS à la dernière valeur (ou échantillon) reçue d'une donnée répond pleinement aux attentes de notre application. Le serveur de simulation dont la principale finalité est de permettre de rejouer la simulation, sauvegardera au niveau applicatif les données nécessaires au rejeu.

Pour les topics périodiques, le paramètre « Deadline » est fixé à la valeur de la période production. Comme il est d'usage pour les applications temps-réel, la valeur du paramètre "Latency Budget" est également fixée à la période de production. Cette valeur est un paramètre système des simulateurs et varie de 20 à 100 ms. En ce qui concerne le paramètre "Reliability", les topics périodiques n'exigent pas de fiabilité absolue puisque même s'il y'a des pertes nous avons une mise à jour régulière. Pour les topics apériodiques, la valeur du paramètre "Reliability" est fixée à "Reliable" puisqu'ils s'agissent des topics transmis soit à l'initialisation soit suite à un évènement ce qui fait que les pertes ne sont pas tolérées vue que la transmission ne se fait qu'une seule fois. Les valeurs du "Latency Budget" sont définies dans le Tableau 3. Elles ont été fournies par ECA Faros.

Le Tableau 3 résume tous les topics tout en décrivant leurs profils de production et leurs valeurs en QoS "Reliability", « Deadline » et "Latency Budget" (La valeur ND = Non Défini). Historiquement, la variable f est un paramètre système des simulateurs qui fixe la fréquence d'envoi des variables périodiques (variables d'état des simulateurs). Cette valeur est fonction de la qualité des écrans utilisés pour afficher les simulateurs dans le monde virtuel et de l'exercice considéré (de 10 à 100 HZ). Pour notre application, cette fréquence varie pour certaines données selon la proximité relative entre simulateurs. Comme indiqué ci-avant, cette fréquence qui conditionne également le délai de distribution des flux périodiques est fonction de la distance relative entre simulateurs.

Nom du Topic	Profil de production	Taille (octets)	direction	BP moyenne Requise (bps)	Reliability	Deadline (ms)	Latency Budget (ms)

CaracterieticquesVehiculeSimule	*Initialisation*	174	Si -> SS -> Si	N.P	Reliable	N.P	N.P
Exercice	*Initialisation*	40	Pi -> SS -> Si	N.P	Reliable	N.P	N.P
ObjetStatique	*Initialisation*	120	SS->Si -> Pi	N.P	Reliable	N.P	N.P
Position	Périodique (1/*f*)	24	Si->Si	880 x *f*	Best-effort	***1/f*** *∈{10, .., 100ms}*	***1/f*** *∈{10, .., 100ms}*
EtatsVehiculeSim	Périodique (1/*f*)	266	Si->Si ->SS ->PI	2816 x *f*	Best-effort	*1/f*	*1/f*
photoScene	Périodique (1/*f*)	453	SS->Si	4312 x *f*	Best-effort	*1/f*	*1/f*
Object3D	Périodique (1/*f*)	201	SS->Si	2296 x *f*	Best-effort	*1/f*	*1/f*
FeuTricolore	Périodique (1/f)	40	SS->Si ->Pi	1008 x *f*	Best-effort	1/f	1/f
VehiculeAUtonome	Périodique (1/*f*)	248	SS->Si -> Pi	2672 x *f*	Best-effort	*1/f*	*1/f*
PhotoRoue3D	Périodique (1/*f*)	49	Si->Si	1080 x *f*	Best-effort	*1/f*	*1/f*
PhotoObjet3D	Périodique (1/*f*)	81	SS->Si	1336 x *f*	Best-effort	*1/f*	*1/f*
EventVehiculeSimule	*Evénementielle*	184	Si->SS -> Si ->PI	N.D	Reliable	N.D	100 ms
Climat	*Evénementielle*	57	Pi->Si ->SS	N.D	Reliable	N.D	100 ms
Scene	*Evénementielle*	50	Pi->Si ->SS	N.D	Reliable	N.D	100 ms
Erreur	*Evénementielle*	125	SS->Pi	N.D	Reliable	N.D	300 ms
Contrôle	*Evénementielle*	116	Pi->SS ->Si	N.D	Reliable	N.D	100 ms

Si : Simulateur physique ; SS : Serveur de Simulation, Pi: Poste Instructeur; N.D. : non défini, N.P : Non pertinent
La taille des structures de données et la bande passante requise sont calculées pour une architecture à 32 bits. La bande passante requise est calculée au niveau de la couche liaison de données (intègre le sur-débit protocolaire de RTPS, UDP, IP et Ethernet. Pour RTPS, le sur-débit comptabilisé est fixé à 40 octets (qui s'applique dans un contexte IPv4 et en supposant l'absence de fragmentation au niveau DDS) [17]

Tableau 3 - Topics DDS de l'application de simulation distribuée considérée

4.2.3 COMPOSANTE DYNAMIQUE DE L'APPLICATION CONSIDÉRÉE

L'aspect dynamique de l'application considérée concerne l'évolution des abonnements/désabonnements (et des besoins en QoS associés) aux données transportant les variables d'état échangées entre les simulateurs. Ceux-ci doivent s'échanger en plus de leur position, leurs informations d'états, à une certaine fréquence (notée *f* ci-avant) et avec un délai plus ou moins contraignant, selon leur distance relative. En effet la logique applicative suppose que, plus les simulateurs sont proches, plus ils peuvent mutuellement influencer le comportement des apprenants. Ainsi, afin de restituer au mieux un rendu réaliste, une fine précision de leurs informations d'état les plus importantes est requise dans les plus brefs délais pour permettre aux apprenants d'être rapidement mis en situation et d'exécuter les actions recommandées par l'apprentissage. C'est ce mode de consommation de ces informations d'état qui confère à cette application le caractère dynamique dans la mesure où les exigences d'acheminement liées à cet échange évoluent au cours du temps en fonction de la distance inter simulateur.

Faute d'expertise métier sur la manière de choisir la fréquence de distribution en fonction de la distance entre simulateurs, les choix présentés dans ce paragraphe ne seront pas motivés. Elles correspondent à une mise en situation des apprenants en milieu urbain avec comme hypothèse sous-jacente que les véhicules simulés auront des difficultés pour dépasser une vitesse de 100Km/h. Le tableau explicite la fonction que nous utilisons pour déduire les besoins en QoS des

flux de données relatifs aux variables d'état des simulateurs. Ces derniers sont définis par les topics DDS : *Position*, *EtatsVehiculeSim*, *VehiculeAUtonome* et *PhotoRoue3D*. Au delà de 50 m de distance, les simulateurs ne s'échangent plus leurs variables d'état (seul le topic *Position* est échangé à une fréquence *f'* de 10Hz). Les autres données périodiques seront transmises à une fréquence fixe de 10 Hz. Pour les variables événementielles, elles seront supportées au niveau réseau par un même lien virtuel avec des besoins de QoS statiques.

Distance relative	**]0 à 25 m]**	**]25m à 50m]**
Fréquence de production *f* (paramètre DDS : Deadline)	40 Hz	20 Hz
Délai de distribution maximal (paramètre DDS : Latency_budget)	25 ms	50 ms

Tableau 4 – caractérisation de la dynamique de l'application

4.3 Infrastructure réseau considérée

L'infrastructure réseau considérée pour ce démonstrateur reprend le réseau du campus du LAAS. Ce choix est justifié par nos précédentes discussions avec ECA Faros qui destinait leur offre de formation à la conduite de groupe avec simulateurs de véhicule à la gendarmerie nationale (ou équivalent étranger). Les simulateurs sont typiquement déployés sur un site appartenant à la gendarmerie ou même sur plusieurs sites reliés via un réseau privé d'entreprise. C'est pour se rapprocher du premier cas (mono-site avec une taille d'un réseau de campus équivalent à celui du LAAS) et la volonté d'avoir une topologie réseau réaliste qui nous amène à l'infrastructure réseau de la Figure 22. Nous retrouvons une topologie hiérarchique avec redondance, classique des réseaux d'entreprise.

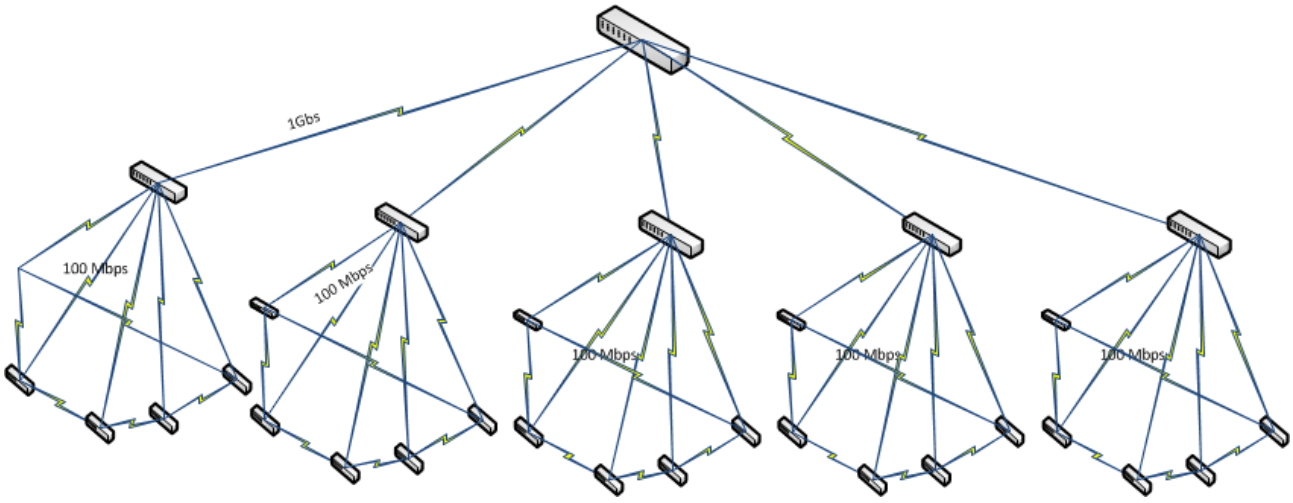


Figure 22 – Topologie du réseau du LAAS

Le démonstrateur n'utilisera pas le réseau du LAAS mais ce dernier sera émulé en utilisant l'émulateur « Mininet » [18] connecté à trois commutateurs Openflow Pica8 P-3295 [19]. Pour les nœuds émulés avec « Mininet », ils seront de type OFSoftSwitch13 [20] car ils implémentent les « meters » Openflow ainsi que les groupes Openflow de type « select » et « all ». OFSoftSwitch13 offre également un support expérimental à Openflow 1.4 avec la possibilité d'utiliser les

« Bundles » Openflow. Les caractéristiques évoquées ci-avant sont des prérequis pour l'algorithme du « VNET Deployer ». Au moment du démarrage des développements (juillet 2015), OFSoftSwitch13 était le seul commutateur logiciel Openflow à les supporter.

4.4 Scénario applicatif et résultats obtenus

4.4.1 DESCRIPTION DU SCÉNARIO APPLICATIF

L'objectif du scénario considéré pour le premier démonstrateur ADN est d'illustrer le fonctionnement du réseau ADN, notamment, en cas de changement de besoins applicatifs. Quatre cas de figure illustrant ces changements seront considérés : rajout de flux applicatifs, changement des besoins en QdS de flux applicatifs vers des garanties plus strictes, changement des besoins en QdS à la baisse et le retrait de flux.

Le scénario repose sur le plan routier de la Figure 23 avec 3 simulateurs de véhicule S1, S2 et S3 qui se déplacent tout au long de l'exécution du scénario en cortège. Comme décrit ci-après, un quatrième simulateur S4 viendra à la rencontre du cortège, parcourra un petit trajet à leur proximité avant de s'en éloigner. Tout au long de l'exécution du scénario, les flux de données échangés entre les simulateurs S1, S2, S3 auront des besoins inchangés. Il en est de même pour les flux de données qu'ils échangent avec le serveur de simulation et le poste instructeur. La dynamique de l'application concernera les flux de données échangées entre le simulateur S4 et les simulateurs du cortège.

Figure 23 – Illustration du scénario à l'instant t0

A l'instant t0, le simulateur S4 est à une distance supérieure à 50 mètres du cortège. S4 n'est pas abonné aux variables d'état des simulateurs S1, S2 et S3 (et inversement). Le Tableau 5 liste les flux de données à cet instant ainsi que leurs exigences de QdS.

Nom du Topic	direction	Nbre de leins virtuels	BP moyenne Requise (bps)	Latency Budget (ms)
Position	Si->Sj {i,j :1..4, i≠j}	4 flux point-multipoints	880 x *f'(f'=100Hz)*	100 ms
EtatsVehiculeSim	Si->Sj {i,j :1..3, i≠j} ->SS	4 flux point-multipoints	2816 x *f (f=40Hz)*	25 ms

	->Pi S4 ->SS -> Pi		2816 x *f* *(f=100Hz)*	100ms
photoScene	SS->Si {i : 1..4}	1 flux point-multipoints	4312 x *f'* *(f'=100Hz)*	100 ms
Object3D	SS->Si {i : 1..4}	1 flux point-multipoints	2296 x *f'*	100 ms
FeuTricolore	SS->Si {i 1..4} ->Pi	1 flux point-multipoints	1008 x *f'*	100 ms
PhotoRoue 3D	Si->Sj {i,j :1..3, i≠j}	3 flux point-multipoints	1080 x *f* *(f=40Hz)*	25 ms
PhotoObjet 3D	SS->Si {i : 1..4}	1 flux point-multipoints	1336 x *f'*	100 ms
EventVehiculeSimule	Si->SS {i,j :1..3, i≠j} -> Sj ->Pi S4 ->SS -> Pi	4 flux point-multipoints	N.D (choix : 100 kbps)	100 ms
Climat	Pi->Si ->SS	1 flux point-multipoints	N.D (choix : 50 kbps)	100 ms
Scene	Pi->Si ->SS	1 flux point-multipoints	N.D (choix : 50 kbps)	100 ms
Contrôle	Pi->Si ->SS	1 flux point-multipoints	N.D (choix : 100 kbps)	100 ms
Erreur	SS->Pi	1 flux point-multipoints	N.D (choix : 100 kbps)	300 ms

Tableau 5 – Liste des flux de données du démonstrateur à l'instant t_0

Une requête VNET englobant l'ensemble des liens virtuels listés ci-avant est donc soumise au réseau ADN. Le composant « Flow aggregator » procède à des agrégations de liens. La version actuelle de ce composant procède à une agrégation lorsque les liens virtuels possèdent la même source et les mêmes destinations. Le résultat de cette dernière est présenté Tableau 6.

Nom du Topic	**Direction**	**Nbre de liens agrégés demandés**	**BP moyenne Requise (bps)**	**Latency Budget (ms)**
Position	Si->Sj {i,j :1..4, i≠j}	4 flux point-multipoints	880 x *f' (f'=100Hz)*	100 ms
EtatsVehiculeSim + EventVehiculeSimule	Si->Sj {i,j :1..3, i≠j} ->SS ->Pi S4 ->SS -> Pi	4 flux point-multipoints	2816 x *f + 100kbps* *(f=40Hz)* 2816 x *f* + *100Kbps* *(f=100Hz)*	25 ms 100 ms
PhotoRoue 3D	Si->Sj {i,j :1..3, i≠j}	3 flux point-multipoints	1080 x *f*	25 ms
photoScene +Objet3D+ PhotoObjet 3D	SS->Si {i : 1..4}	1 flux point-multipoints	7944 x *f'* *(f'=100Hz)*	100 ms
FeuTricolore	SS->Si {i 1..4} ->Pi	1 flux point-multipoints	1008 x *f'*	100 ms
Climat+Scene+Contrôle	Pi->Si ->SS	1 flux point-multipoints	N.D (choix : 200 kbps)	100 ms
Erreur	SS->Pi	1 flux point-multipoints	N.D (choix : 100 kbps)	300 ms

Tableau 6 – Liste des liens agrégés de la requête VNET soumise à l'instant t_0 au « Resource Allocator »

A l'instant t1, le simulateur S4 se trouve à une distance du cortège qui est légèrement inférieure à 50 m (voir Figure 24). Une phase de souscription aux topics « *EtatsVehiculeSim* » et « *PhotoRoue3D* » des simulateurs S1, S2 et S3 s'engage. Similairement, les simulateurs S1, S2 et S3 s'abonnent à leurs

tours aux données des mêmes topics issues de S4 (voir Tableau 8). Cela se traduit au niveau du réseau ADN par la réception :

- d'une requête VNET de rajout d'un lien virtuel correspondant aux données de type « *EtatsVehiculeSim* » produites par S4 et deux mises à jour de liens virtuels en y intégrant de nouvelles destinations relatives aux nœuds réseau de S1, S2, S3 (voir Tableau 7). Ces demandes émanent du « Application Classifier» et sont traités par le « Requets Handler » qui après avoir sollicité le « Flow Aggregator » soumet les requêtes du Tableau 8 à l'allocateur de ressources.

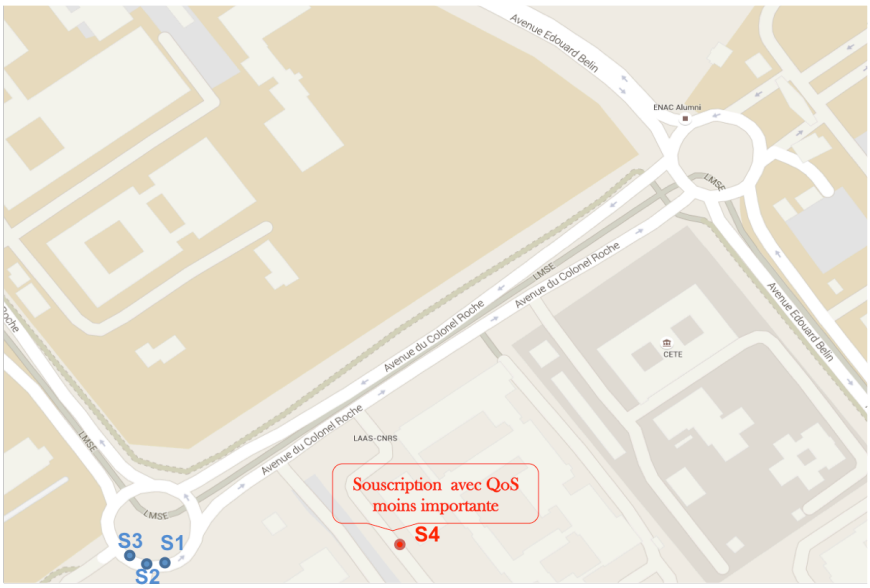


Figure 24 – Illustration du scénario à l'instant t1

Nom du Topic	direction	Nbre de leins virtuels	BP moyenne Requise (bps)	Latency Budget (ms)
EtatsVehiculeSim	S4->Si {i:1..3}	1 flux point-multipoints	2816 x *f* *(f=20Hz)*	50 ms
PhotoRoue3D	**S4->Si {i :1..3}**	1 flux point-multipoints	1080 x *f* *(f=20Hz)*	50 ms
EventVehiculeSimule	**Si- -> Si {i :1..3}** ->SS ->Pi	1 flux point-multipoints	N.D (choix : 100 kbps)	100 ms

Tableau 7 – Liste des requêtes issues du Simulateur S4 à l'instant t_1

Nom du Topic	Direction	Nbre de liens agrégés demandés	BP moyenne Requise (bps)	Latency Budget (ms)
EtatsVehiculeSim + EventVehiculeSimule	S4 -> **Si {i :1..3}** ->SS -> Pi	1 flux point-multipoints	2816 x *f* + *100Kbps* *(f=50Hz)*	50 ms
PhotoRoue3D	**S4->Si {i :1..3}**	1 flux point-multipoints	1080 x *f* *(f=50Hz)*	50 ms

Tableau 8 – Liste des liens agrégés de la requête VNET soumise à l'instant t_1 au « Resource Allocator » pour supporter les données produites par S4

- Similairement, et après l'intervention des composants « Application Classifier » et « Requets Handler », trois requêtes de mise à jour de liens virtuels sont soumises au « Resource Allocator » pour inclure le simulateur S4 comme réceptionnaire des variables d'état des simulateurs S1, S2 et S3.

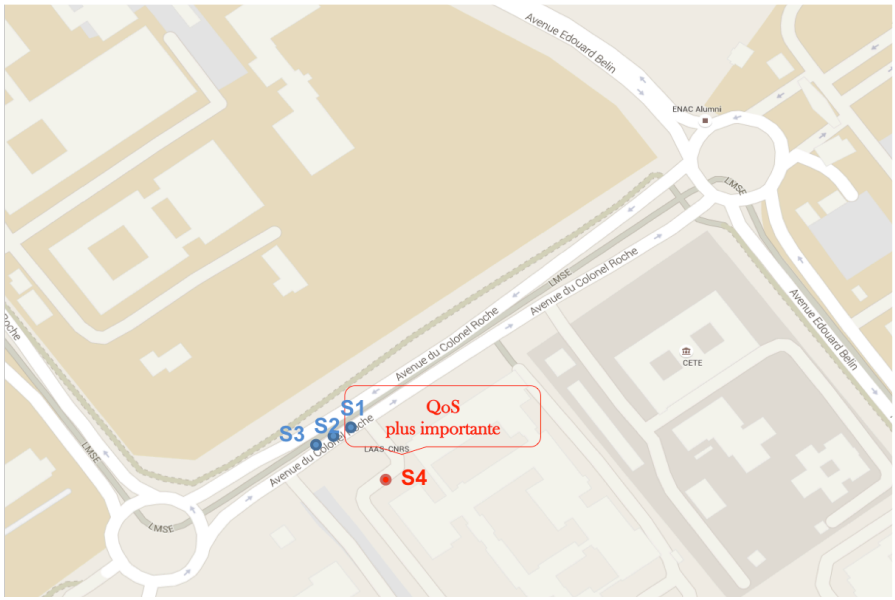


Figure 25 – Illustration du scénario à l'instant t2

A l'instant t2, le simulateur S4 se trouve à une distance du cortège inférieure à 25 m (voir Figure 25). Cela implique un changement de la fréquence avec laquelle S4 échange les variables d'états avec les simulateurs du cortège. Des requêtes de mises à jour sont donc transmises par l' »Application Classifier» pour les mêmes liens que ceux traités à l'instant t1. Les nouvelles exigences de QdS découlent de l'augmentation de la fréquence d'échange des données d'état des simulateurs avec S4.

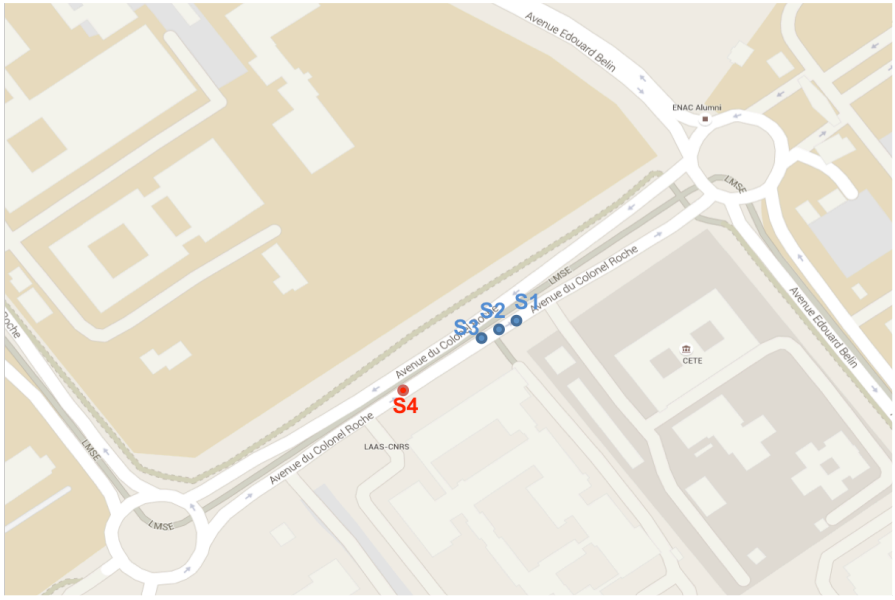


Figure 26 – Illustration du scénario à l'instant t3

A l'instant t3, les besoins de l'application n'ont pas évolué (voir Figure 26). Elles évoluent à l'instant t4 où le simulateur S4 se retrouve à une distance du cortège supérieure à 25m (voir Figure 27). Des requêtes de mises à jour équivalent à l'instant t1 sont soumises au « Request Handler ».

Figure 27 – Illustration du scénario à l'instant t4

A l'instant t5, le simulateur S4 est de nouveau au delà des 50 mètres du cortège. Une phase de désouscription aux données des simulateurs S1, S2 et S3 est déclenchée (et inversement). On se retrouve de nouveau dans une situation équivalente à l'instant t_0.

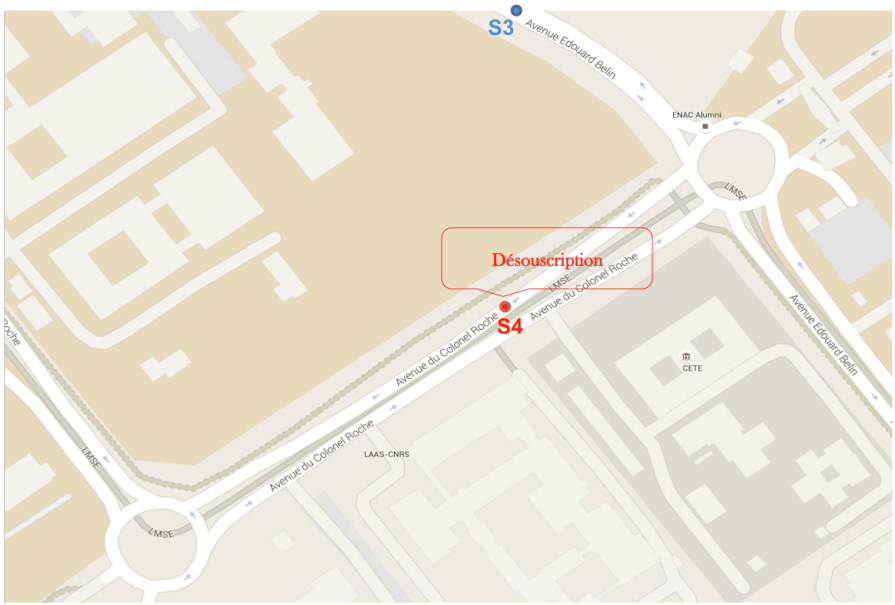


Figure 28 – Illustration du scénario à l'instant final t5

4.4.2 RÉSULTATS DU DÉMONSTRATEUR

Les figures Figure 29 et Figure 30 illustrent le résultat des allocations des ressources sur l'infrastructure réseau considérée pour les différents liens virtuels à provisionner aux instants t_0 (voir Tableau 6) et t_1 (voir Tableau 7).

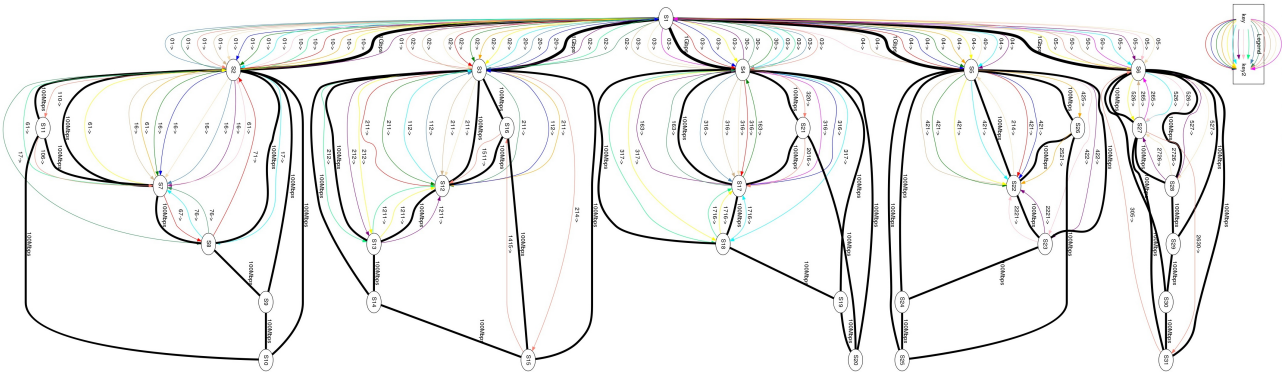


Figure 29 – Résultats des allocations suite aux requêtes VNET soumises à l'instant t0

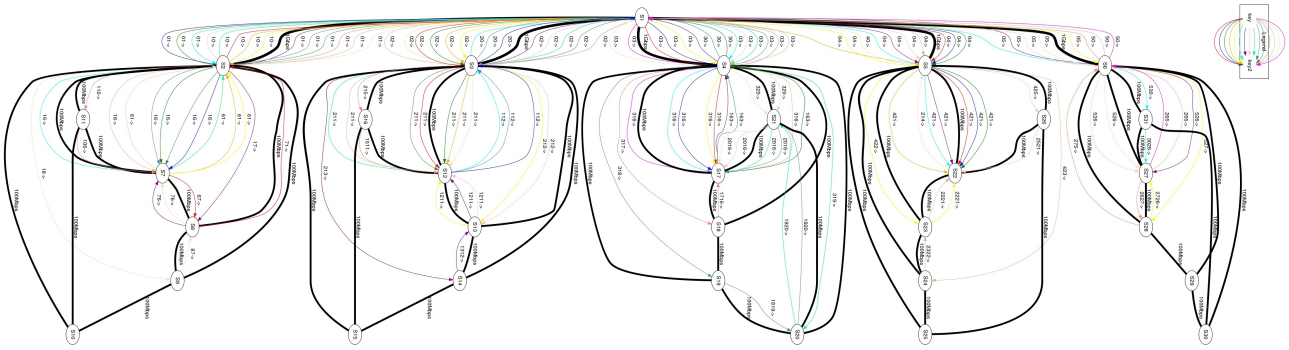


Figure 30 – Résultats des allocations suite aux requêtes VNET soumises à l'instant t1

La Figure 31 illustre de manière plus visuelle les chemins de données alloués par l'algorithme d'allocation de ressources aux liens virtuels qui supportent les échanges relatifs au topic «*EtatsVehiculeSim* » depuis les différents simulateurs S1, S2, S3 et S4. Les chemins de données en vert correspondent aux données produites par le simulateur S1. Les chemins de données en rouge supportent les données produites par le simulateur S4. Les chemins en jaune (respectivement violet) supportent les données produites par le simulateur S2 (resp. S3).

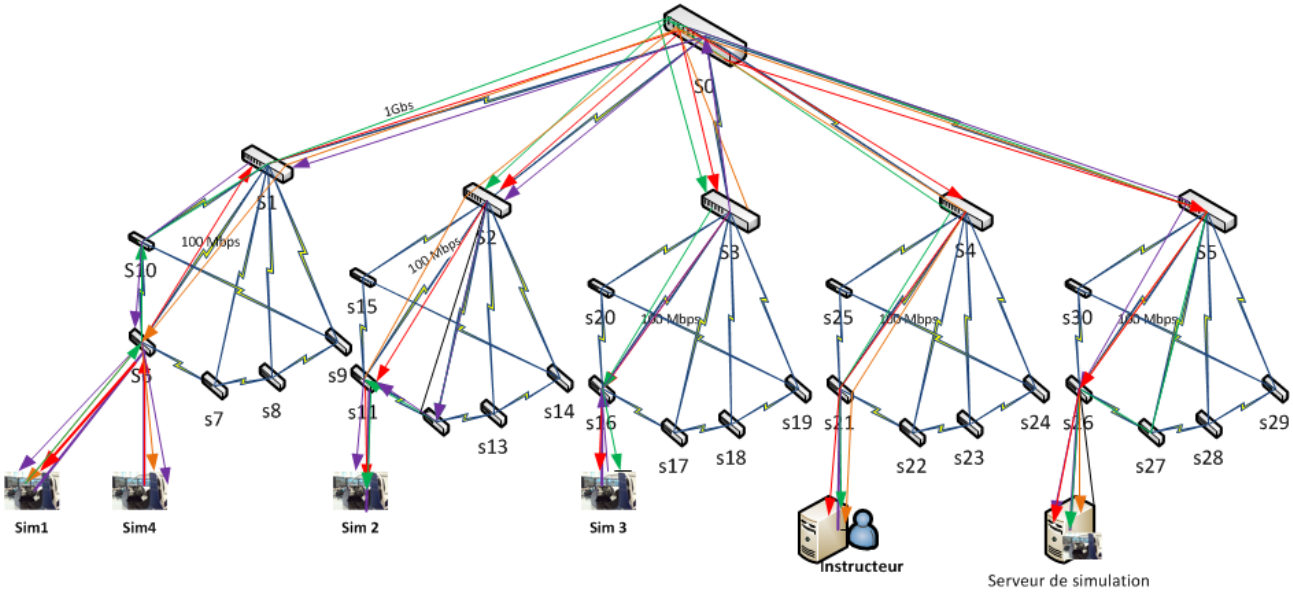


Figure 31- Chemins des données des liens virtuels supportant les échanges relatifs au topic «*EtatsVehiculeSim* » calculés à l'instant t1

4.5 Conclusion

Cette section a décrit le premier démonstrateur du projet ADN qui vise à illustrer le service qu'offre un réseau ADN à une application DDS dynamique dans un contexte réseau maîtrisé (réseau filaire). Ce démonstrateur illustre, notamment, le niveau de finesse pouvant être atteint par l'approche ADN dans la prise en compte des besoins applicatifs. C'est l'un des points caractéristiques de l'approche ADN, même si celle-ci, est également applicable dans un contexte plus général qui n'impose pas une telle description des besoins applicatifs.

Le démonstrateur ainsi décrit est opérationnel même les algorithmes de certains composants sont amenés à être enrichis et améliorés d'ici la fin du projet.

5. Evaluations

5.1 Introduction

Les évaluations qui ont été rálisées ont porté sur l'algorithme d'allocation de ressources. En effet, à défaut d'une infrastructure réseau réelle suffisamment complexe, il ne nous semble pas pertinent dévaluer les performances des algorithmes du composant *VNET Deployer* et *Application Classifier*. Nous prsentons dans cette section certains résultats de l'analyse de performances de cet algorithme sur la topologie réseau de Géant ainsi que sur la topologie réseau de campus de notre laboratoire. Les objectifs de ces différentes analyses sont :

1. La mise en évidence de l'apport (taux d'acceptation et revenus) du modèle par rapport à des méthodes heuristiques.
2. La caractérisation des temps de calcul en fonction du taux d'arrivée des requêtes, de leurs types (unicast / multicast) et de l'utilisation du path-splitting.
3. L'influence des paramètres sur les résultats du modèle (à nouveau en terme de taux d'acceptation et de revenus).
4. La visualisation de l'équilibre des charges dans le réseau via l'utilisation d'une fonction de répartition sur le taux d'utilisation des liens.

5.2 Modèle de simulation considéré

5.2.1 Modèles de réseau

Contrairement aux travaux existants qui expérimentent sur des réseaux générés aléatoirement, le choix a été fait d'utiliser une topologie réseau réelle. Nous utilisons deux modèlles de réseau pour les expérimentations :

— le réseau européen de recherche, GÉANT, qui possèdent des liens reliant la plupart des capitales et villes importantes européennes. La version considérée du réseau contient 41 Routeurs (ou noeuds) et 60 Liens (voir figure 5.1).
— le réseau de campus du LAAS présenté précédemment qui est constitué de 31 noeuds et 30 liens.

5.2.1.1 Capacité des liens

Les capacités des liens du réseau Géant sont fournies par la carte. Pour les liens n'ayant pas une capacité précise, les règles suivantes sont utilisées :

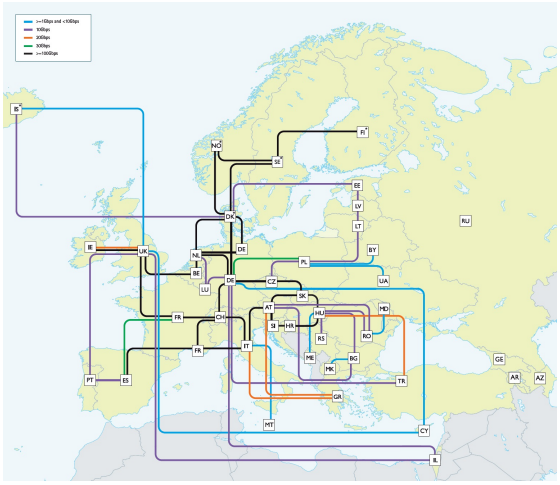


FIGURE 5.1 – Carte de GÉANT

— $\geq$ **1 Gbps,** $<$ **10 Gbps** : La capacité du lien est fixée à 5 Gbps.
— $\leq$ **100 Gbps** : La capacité du lien est ramenée à 100 Gbps.

Pour le réseau de campus, les liens d'éxtrémité sont fixés à $100Mbps$ contre $1Gbps$ pour l'épine dorsale du réseau.

5.2.1.2 Les délais de transmission des liens

La carte du réseau Géant ne fournit pas directement les délais de transmissions des liens, mais ceux-ci sont calculables en utilisant la localisation des routeurs. Les liens sont considérés en ligne droite d'un routeur à l'autre, et la vitesse de transmission v_{tr} est fixée à 3.10^8 m/s. La distance terrestre $D(a, b)$ entre deux routeurs est calculée à partir de leurs longitudes et latitudes, le délai de propagation du routeur a vers le routeur b, $tr(a, b)$, est alors obtenu simplement par :

$$tr(a, b) = \frac{D(a, b)}{v_{tr}} \tag{5.1}$$

Pour le réseau de campus, les délais de propagation sont considérés nuls.

5.2.1.3 La capacité des routeurs

La capacité de commutation des routeurs, exprimées en nombre d'entrées disponibles dans la table des flots, est fixée à 2000 entrées. On considère ici qu'une partie des entrées de la table des flux des noeuds sont utilisées pour le provisionnement des VNETs, le reste étant utilisé pour d'autres fonction réseau. La table de groupes Openflow est fixée à 512 entrées.

5.2.2 Modèles de charge

5.2.2.1 Le taux d'arrivée des requêtes

Les requêtes arrivent selon un processus de Poisson de paramètre λ, avec λ variant de 4 à 10 requêtes toutes les 100 unités de temps par pas de 1 : $\lambda \in \{0.04, 0.05, 0.06, 0.07, 0.08, 0.09, 1.0\}$.

5.2.2.2 La durée de vie

Les requêtes possèdent une durée de vie moyenne de 1000 unités de temps suivant une loi exponentielle de paramètre $\mu = 1/1000$.

5.2.2.3 Les différents types de requêtes

Deux types de requêtes ont été considérés pour les expérimentations : Unicast et Multicast.

5.2.2.3.1 Les requêtes Unicast Les requêtes unicast contiennent un seul lien virtuel, soit un VNET avec une seule destination. Les paramètres du lien virtuel sont générés aléatoirement selon des lois uniformes, avec les contraintes suivantes :

Bande passante Entre 7 et 12 Gbps (respectivement 1 à $3Mbps$) pour les expérimentations relatives à Géant (resp. Réseau de campus)

Délai Entre 50 et 80 millisecondes pour les expérimentations relatives à Géant et entre 20 et $200ms$ pour celles relatives au Réseau de campus.

5.2.2.3.2 Les requêtes Multicast Les requêtes multicast contiennent plusieurs liens virtuels point à multipoint : Entre 4 et 6 liens virtuels, chacun possédant entre 4 et 6 destinations. Les paramètres de chaque lien virtuel sont générés aléatoirement selon des lois uniformes, avec les contraintes suivantes :

Bande passante Entre 200 Mbps et 3 Gbps (respectivement 1 à $3Mbps$) pour les expérimentations relatives à Géant (resp. Réseau de campus)

Délai Entre 50 et 80 millisecondes pour les expérimentations relatives à Géant et entre 20 et $200ms$ pour celles relatives au Réseau de campus.

Remarque : Pour qu'une requête VNET contenant plusieurs liens virtuels soit acceptée par le modèle, il faut que tous ses liens puissent être alloués sur le réseau, i.e. s'il n'existe pas de chemin de capacité suffisante reliant la source et les destinations d'un lien, la requête entière est rejetée.

5.2.2.4 Le path-splitting

Les expérimentations ont été effectuées avec utilisation du path-splitting ($PS_{ratio} \in \{10\%, 33\%, 45\%\}$) et sans ($PS_{ratio} = 100\%$). Un lien virtuel avec un PS_{ratio} de 10% (resp. 33% et 45%) ne pourra être allouée sur plus de 10 (resp. 3 et 2) chemins différents, caque chemin étant utilisé pour 10% (resp. 33% et 45% de la bande passante). Avec un $PS_{ratio} > 50\%$ (100% dans notre cas), le lien doit être alloué sur un seul chemin physique.

5.2.2.5 Le choix des sources et destinations

Les noeuds sources et destinations des liens virtuels sont choisis suivant une probabilité proportionnelle à la somme des capacités des liens auxquels ils sont connectés, soit :

$$p_i \propto \sum_{(i,j)\in E} c_{ij}$$

5.2.3 Les paramètres d'expérimentation

5.2.3.1 La durée d'expérimentation

Les expérimentations ont toutes été effectuées sur une durée de 10000 unités de temps. La courbe 5.5 montre que cette durée est suffisante pour atteindre un état stationnaire du système.

5.2.3.2 La fonction objective

La fonction objective utilisée est celle définie [1], avec la variation numéro 3 (variation utilisée dans la présentation du modèle).

5.2.3.2.1 Les paramètres α, β et γ Les coefficients α_1 et α_2 ont été fusionnés et fixés à la valeur 1 : $\alpha_1 = \alpha_2 = \alpha = 1$.

Les coefficients β_1 et β_2 ont également été fusionnés. 4 valeurs différentes ont été utilisées pour les expérimentations : $\beta_1 = \beta_2 = \beta \in \{1, 50, 100, 150\}$.

La minimisation des délais n'étant pas recherchée dans les expérimentations, le coefficient γ a été fixé à la valeur 0.

5.2.4 Fonctionnement des expérimentations

On considérera dans la suite des instances de test : une instance est un ensemble de requêtes, générées sur une durée fixe (10000 unités de temps dans notre cas) avec un taux d'arrivée et des paramètres définis.

Pour chaque type de requête (unicast et multicast), et chaque taux d'arrivée (de 4 à 10 requêtes toutes les 100 unités de temps), deux instances ont été générées avec des paramètres β et PS_{ratio} fixés, soit un total de $2 * 2 * 7 = 28$ instances.

Chaque instance générée a ensuite été rejouée en faisant varier les paramètres β et PS_{ratio}, portant le total d'instance à $28 * 16 = 448$.

Chaque instance peut alors être caractérisée ainsi :

$$(Type, \lambda, \beta, PS_{ratio})$$

5.3 Métriques de performance

Les performances de l'algorithme ont été mesurées à l'aide de plusieurs métriques : le taux d'acceptation des requêtes, les « revenus » générés, le taux d'utilisation des liens et la charge globale des liens.

Les métriques ont été mesurées :

— À la fin de la simulation (à $T = 10000$ unités de temps), les graphes sont alors fournis en fonction du taux d'arrivée des requêtes λ.
— Sur toute la période de simulation d'une expérience liée à une instance $(Type, \lambda, \beta, PS_{ratio})$, les graphes sont alors fournis en fonction du temps de simulation T (en unité de temps).

5.3.1 Le taux d'acceptation des requêtes

Le taux d'acceptation des requêtes correspond simplement au nombre de requêtes qui ont été allouées sur le réseau, par rapport au nombre total de requêtes générées pendant la durée d'une simulation (ou une partie de cette durée dans le cas d'étude sur la période de simulation).

$$\tau_{acceptation} = \frac{\text{Nombre de requêtes allouées}}{\text{Nombre de requêtes reçues}}$$

Exemple : Si une instance contient 400 requêtes, et que 260 requêtes ont été allouées avec succès, alors le taux d'acceptation est de $\tau = \frac{260}{400} = 65\%$.

5.3.2 Les revenus générés

On considère que chaque requête génère des revenus pour le fournisseur de réseau virtuel si son allocation sur le réseau réussi. Le revenu généré par une requête est calculé par la formule ci-dessous :

$$Revenu(R) = \begin{cases} \mu(R) * \sum\limits_{k \in K} \left(b_k * \sqrt{T_k}\right) & si \text{ R a été allouée} \\ 0 & si \text{ R n'a pas pu être allouée} \end{cases}$$

Avec $\mu(R)$ la durée de vie de la requête (en unités de temps), K le nombre de sessions de la requête et b_k et T_k la bande passante demandée (en Mbps) et le nombre de destinations de la session k. On utilise $\sqrt{T_k}$ au lieu de T_k directement afin de se rapprocher des revenus réels : du point de vue du consommateur, un lien multicast vers N destinations est moins polyvalent que N liens unicast, leurs coûts ne peuvent donc pas être identique.

Exemple : Une requête durant 800 unités de temps composée d'une session avec 3 destinations pour une bande passante de 400Mbps, et une session avec 2 destinations pour une bande passante de 1Gbps, aura un revenu de :

$$Revenu(R) = 800 * (400 * \sqrt{3} + 1000 * \sqrt{2}) = 800 * (693 + 1414) = 1,685.10^6$$

Le revenu global de la simulation est alors calculé comme la somme du revenu de toutes les requêtes allouées.

$$Revenu(Simulation) = \sum Revenu(R)$$

5.3.3 La charge globale des liens du réseau

La charge globale du réseau correspond à la quantité totale de ressources consommées sur les liens (à un instant donné) par rapport à la capacité totale des liens, soit :

$$C_{liens} = \frac{\sum\limits_{e \in E} \overline{c_e}}{\sum\limits_{e \in E} c_e}$$

Remarque : Comme on considère des liens full-duplex, il est important de noter que l'ensemble E des liens contient, pour chaque lien de GÉANT, deux liens : Un de i vers j et un de j vers i. Lors du calcul de la charge globale des liens du réseaux, chaque lien de GÉANT est donc considéré deux fois, i.e. pour que la charge soit de 100%, il faudrait que chaque lien utilise la totalité de ses ressources dans les deux sens.

5.3.4 Le taux d'utilisation des liens

Le taux d'utilisation des liens correspond à la moyenne du taux d'utilisation de chaque lien (à un instant donné), c'est à dire les ressources consommées sur le lien par rapport aux ressources disponibles, soit :

$$\tau_{liens} = \frac{1}{|E|} * \sum_{e \in E} \frac{\overline{c_e}}{c_e}$$

Avec c_e la capacité du lien e et $\overline{c_e}$ les ressources consommées sur le lien e à l'instant considéré. Comme $\forall e \in E,\ \overline{c_e} \leq c_e$, $\tau_{liens} \in [0, 1]$.

Remarque : Tout comme pour la charge globale des liens du réseau, il est important de noter que pour atteindre un taux d'utilisation de 100%, il faudrait que chaque lien utilise 100% de ses ressources dans les deux sens.

Tous les liens n'ont pas la même capacité donc $\tau_{liens} \neq C_{liens}$. Contrairement à la charge du réseau, le taux d'utilisation des liens ne fait pas la différence entre un lien de capacité 50 Gbps chargé à 80% et un lien de capacité 4 Gbps chargé à 80%.

La charge globale des liens du réseau et le taux d'utilisation peuvent être utilisés comme indicateur de l'équilibre du réseau. Les résultats concernant la charge sont étudiés dans la section 5.4.4.3, et des fonctions de répartition du taux d'utilisation des liens en fin de simulation sont fournies dans la section 5.4.5.

5.4 Résultats expérimentaux sur le réseau Géant

5.4.1 Comparaison avec des méthodes heuristiques

La méthode présentée dans ce document a été comparée à différentes heuristiques de Plus Court Chemin (PCC). En effet, les algorithmes PCC sont couramment utilisés pour le routage de paquets dans les réseaux actuels, et peuvent donc s'adapter facilement pour de l'allocation de liens virtuels.

Pour une requête unicast, l'algorithme de PCC utilisé est standard (avec les coûts définis ci-dessous). Pour une requête multicast, la procédure adoptée est la suivante :

1. L'allocation des liens virtuels se fait les uns à la suite des autres. La requête est rejetée si, pour un des liens, le chemin trouvé ne possède pas une capacité suffisante.
2. Pour les liens multicast, chaque chemin $(source, destination)$ est calculé par l'algorithme PCC de façon isolée. L'allocation finale du lien est ensuite calculée par le max sur chaque lien physique des capacités allouées aux liens logiques (procédure semblable à celle utilisée par notre modèle).

Trois versions du calcul du coût d'un lien ont été considérées :

Coût = 1 Chaque lien possède un coût de 1, cela revient à chercher le chemin passant par le moins de routeurs possibles.

Coût $= f(c_e)$ Le coût d'un chemin est inversement proportionnel à sa capacité (Coût $= 1/c_e$). C'est le calcul de coût utilisé dans le protocole OSPF.

Coût $= g(c_e, \overline{c_e})$ Le coût d'un chemin est inversement proportionnel aux ressources disponibles sur le lien (Coût $= \frac{1}{\overline{c_e} - c_e}$).

Les graphiques de la figure 5.2 présentent les résultats de comparaison de notre méthode avec les algorithmes précédemment décrits. Les paramètres $\beta = 150$ et $PS_{ratio} = 10$ ont été choisis suite aux résultats comparatifs présentées dans la partie 5.4.4.

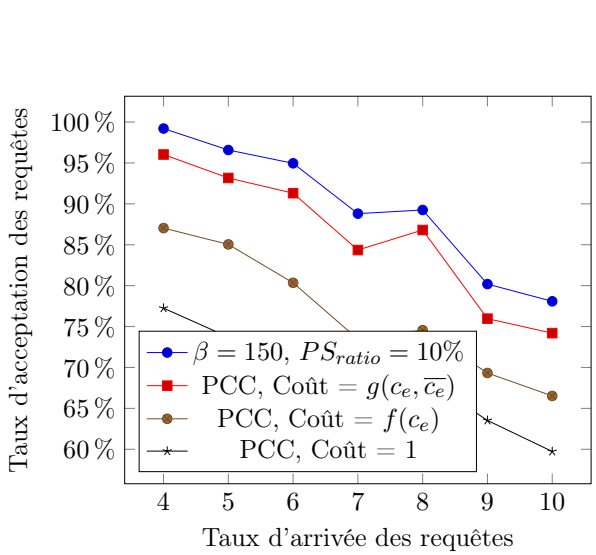
Les courbes montrent que quelque soit le type d'instance considérée, notre algorithme présente un apport notable en terme de taux d'acceptation et surtout de revenu. L'augmentation est plus importante sur des instances de type multicast (2% pour des taux d'arrivée élevé jusqu'à 10% pour un taux d'arrivée de 0.04) que sur des instances de type unicast (entre 3 et 4%).

L'apport de la méthode est d'autant plus visible sur les revenus : Jusqu'à 300 Tb pour les instances de type unicast et 600 Tb pour celles de type multicast par rapport à la meilleure des heuristiques.

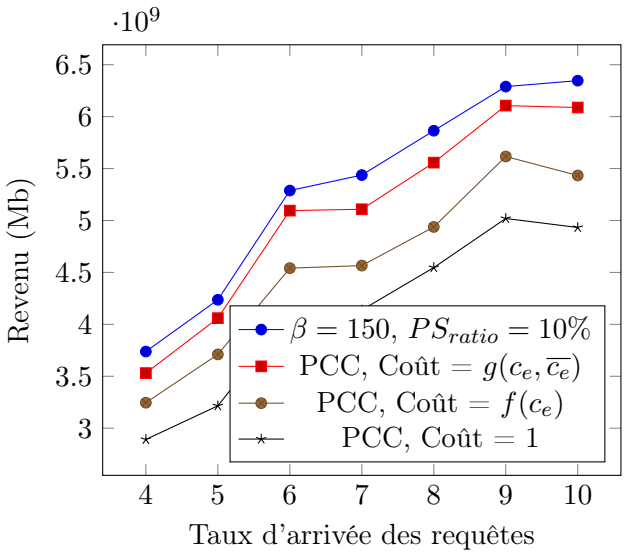
5.4.2 Caractérisation des temps de calcul

Les courbes 5.3 et 5.4 indiquent le temps moyen et maximal d'allocation des requêtes sur le réseau. Les valeurs fournies sont calculées sur l'ensemble des requêtes acceptées en utilisant les temps CPUs retournés par le solveur CPLEX [1].

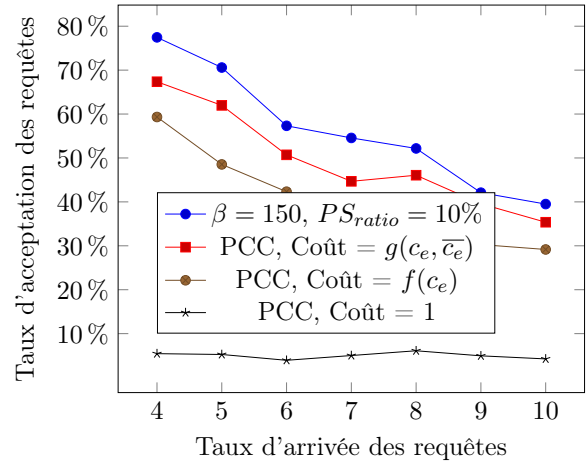
1. `http://pic.dhe.ibm.com/infocenter/cosinfoc/v12r4/index.jsp?topic=%2Filog.odms.ide.help%2Frefcppopl%2Fhtml%2Fclasses%2FIloTimer.html`



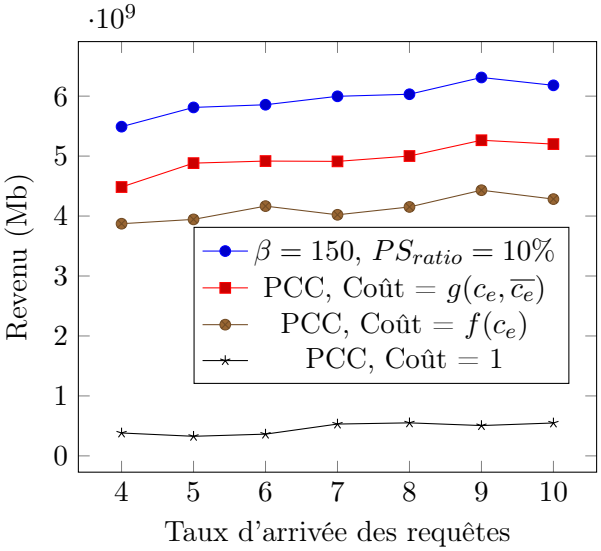
(a) Requêtes de type Unicast



(b) Requêtes de type Unicast



(c) Requêtes de type Multicast



(d) Requêtes de type Multicast

FIGURE 5.2 – Comparaison de différents algorithmes d'allocation

Sur les instances de type unicast, les temps de calculs sont en moyenne faibles (inférieurs à 40 millisecondes sans path-splitting, et à 100 ms avec path-splitting) et ne dépassent jamais 700 millisecondes (la limite de 15 secondes fixées pour le solveur n'est jamais atteinte). Pour les instances de type multicast, les temps de calcul sont plus élevés (entre une et quatre secondes en moyenne) et la limite du solveur est atteinte (les maximums dépassent les 15 secondes pour tous les taux d'arrivée). Cette différence s'explique facilement lorsque l'on compare le nombres de variables et contraintes nécessaires pour la modélisation des requêtes de type multicast et unicast.

Quelque soit le type d'instance considéré, l'utilisation du path-splitting augmente de façon notable le temps de calcul, bien que celui-ci reste largement acceptable dans un concept de fournisseur de réseaux virtuels (les temps de calcul restent négligeables en comparaison des temps de déploiement qui peuvent atteindre plusieurs minutes).

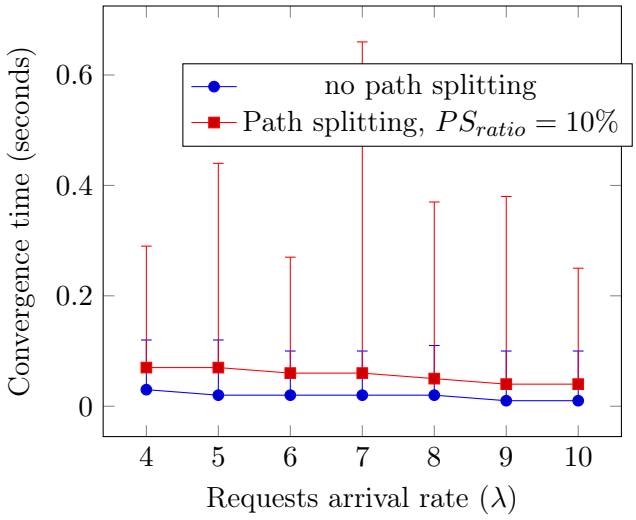


FIGURE 5.3 – Convergence time - Unicast requests

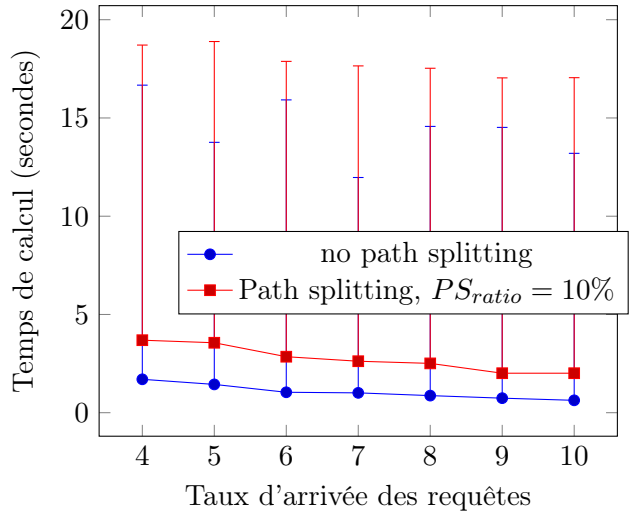


FIGURE 5.4 – Convergence time - Multicast requests, $\beta = 100$

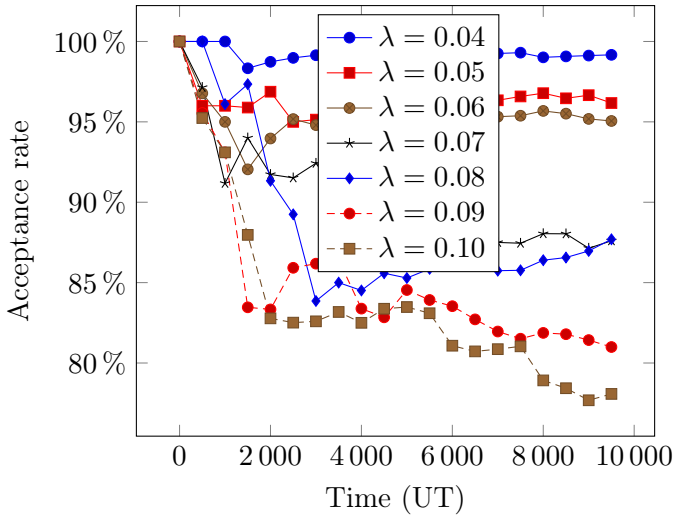
L'influence du path-splitting sur le temps de calcul est directement liée à l'espace de recherche des variables de flux f_t^k - Dans un scénario sans path-splitting, elles sont limitées à deux valeurs possibles (0 ou b_k), ce qui n'est plus le cas à partir du moment où on utilise un PS_{ratio} inférieur à 50%.

5.4.3 Validation de la durée de simulation

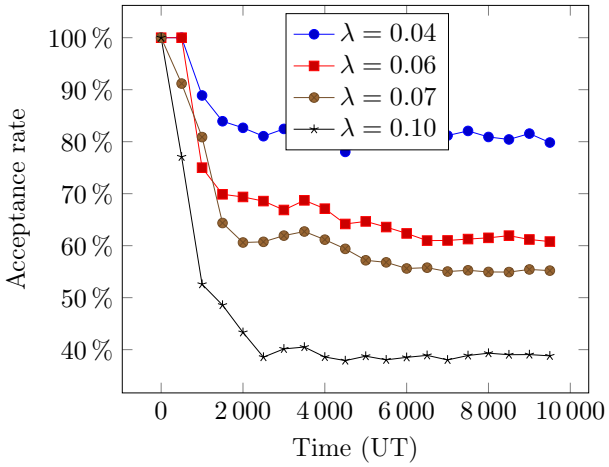
Les courbes de la figure 5.5 montrent l'évolution du taux d'acceptation et du taux d'utilisation des liens du réseau au cours de la simulation.

À la fin de la simulation ($T = 10000$ unités de temps), le système a atteint un régime stationnaire. Ce résultat nous permet de cibler les analyses suivantes en regardant uniquement les valeurs en fin de simulation.

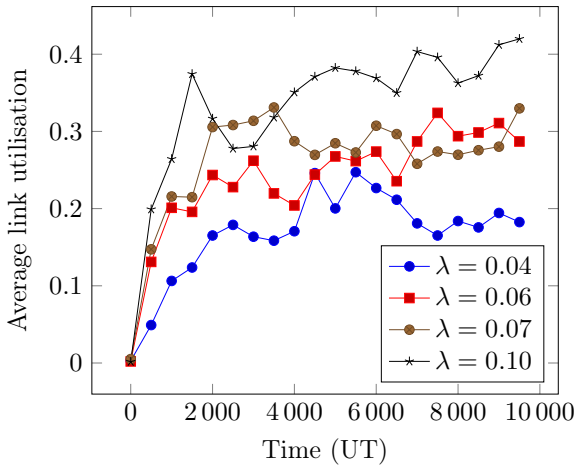
Sur les courbes 5.5a et 5.5b, on remarque que les instances composées de requêtes de type multicast se stabilisent beaucoup plus vite (vers $T = 2000$ unités de temps) que



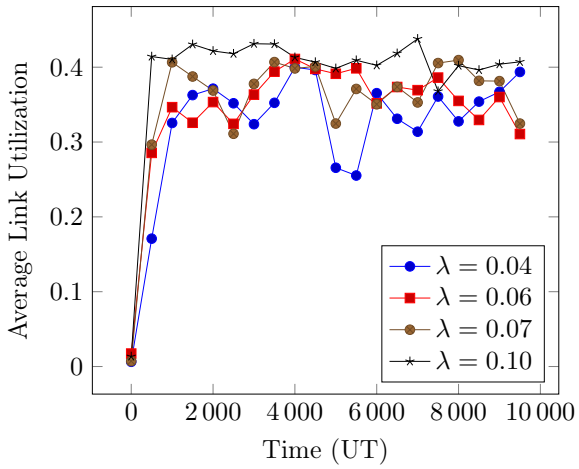
(a) Taux d'utilisation des liens - Requêtes Unicast



(b) Taux d'utilisation des liens - Requêtes Multicast



(c) Taux d'acceptation - Requêtes Unicast



(d) Taux d'acceptation - Requêtes Multicast

FIGURE 5.5 – Taux d'acceptation et d'utilisation des liens en fonction du temps pour des requêtes Multicast et Unicast - $\beta = 150$, $PS_{ratio} = 0.45$

les requêtes de type unicast. La chute brutale du taux d'acceptation observée sur ces courbes s'explique par l'augmentation brutale et la stabilisation du taux d'utilisation des liens, visible sur les graphiques 5.5c et 5.5d.

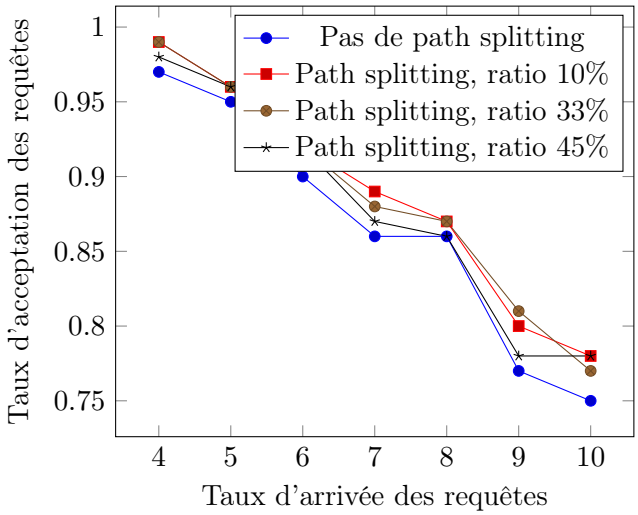
Contrairement aux instances composées de requêtes de type unicast (graphiques 5.5c), les instances composées de requêtes de type multicast atteignent toutes le même taux d'utilisation des liens, quelque soit leur taux d'arrivée. Même pour des taux d'arrivée faibles ($\lambda = 0.04$), le réseau semble saturer rapidement, ainsi le nombre maximal de requêtes pouvant être allouées en même temps sur le réseau est très vite atteint.

5.4.4 Influence des paramètres du modèle

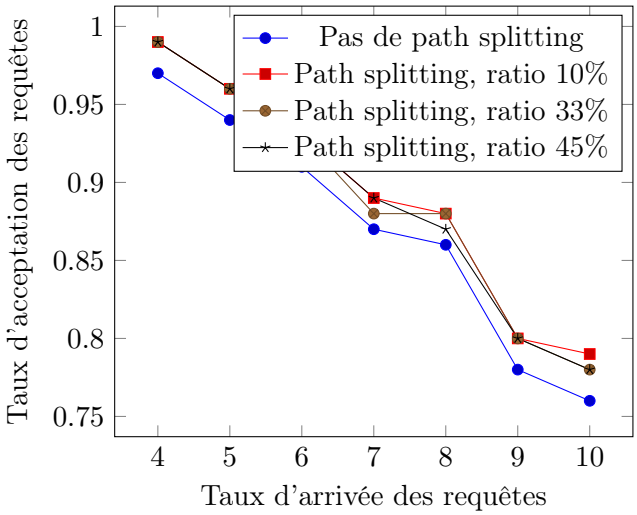
Dans cette section, nous allons étudier l'influence respective des paramètres PS_{ratio} et β (prise en compte de la charge des routeurs) de notre modèle.

5.4.4.1 Taux d'acceptation des requêtes

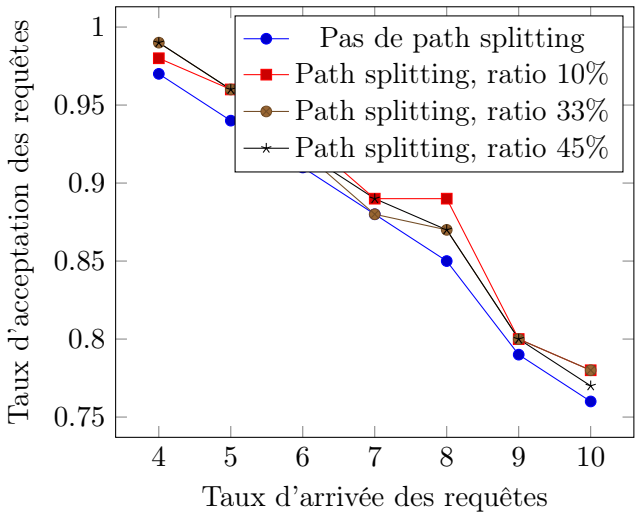
Les courbes 5.6a à 5.6d et 5.7a à 5.7d présentent les taux d'acceptation des requêtes pour différentes instances de type unicast et multicast.



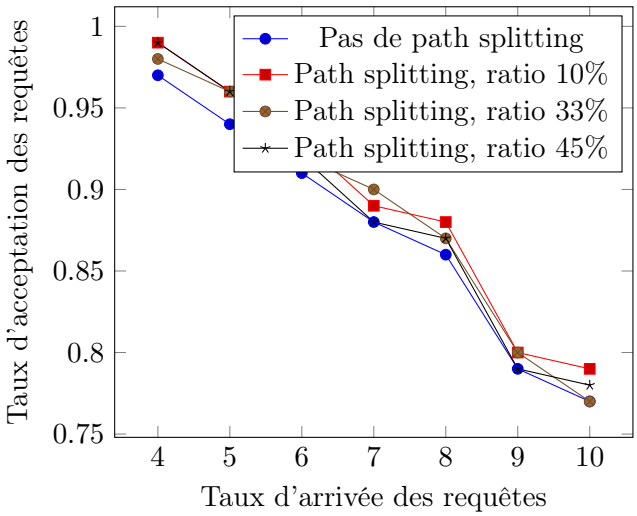
(a) Taux d'arrivée - Béta = 1



(b) Taux d'arrivée - Béta = 50



(c) Taux d'arrivée - Béta = 100

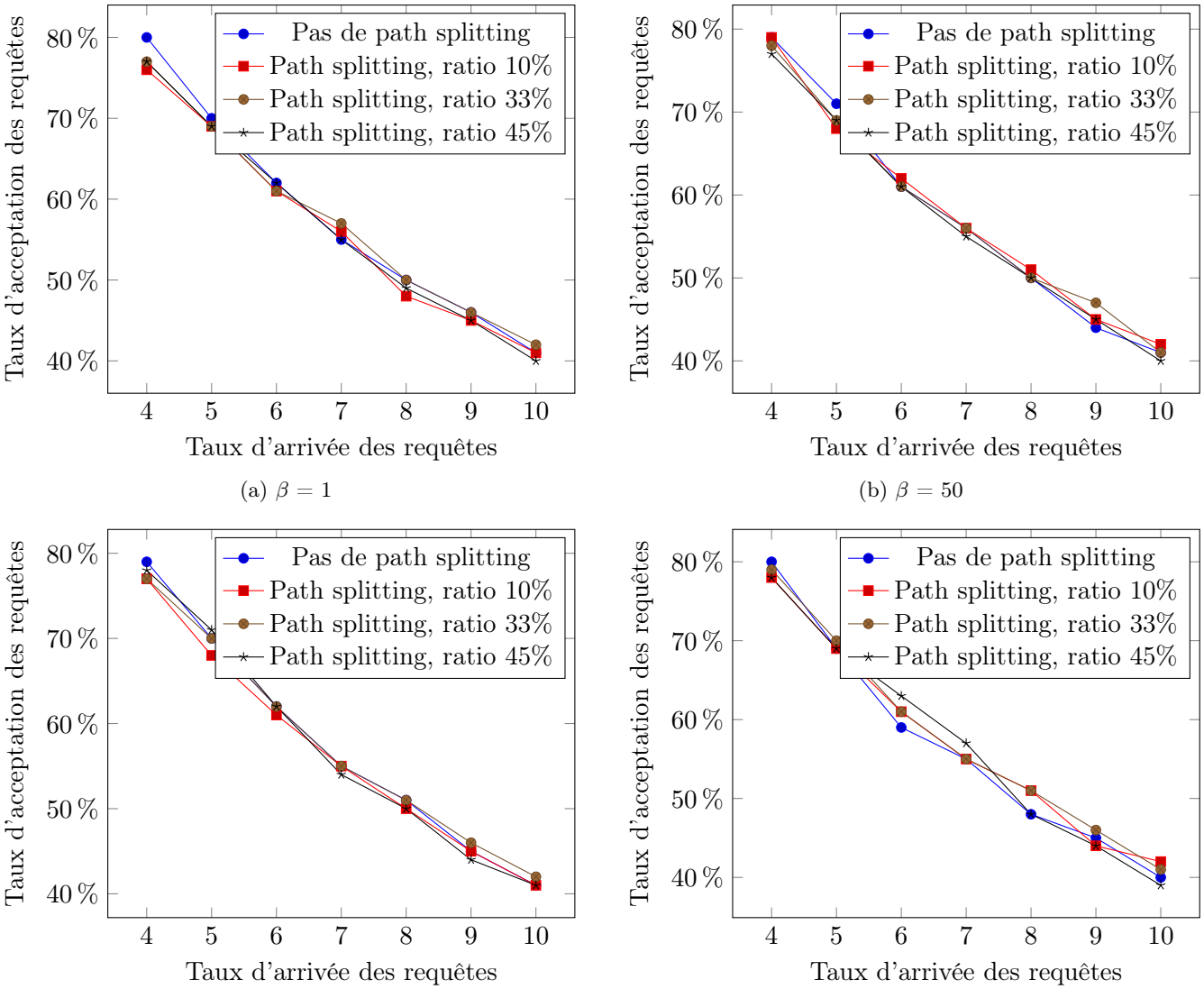


(d) Taux d'arrivée - Béta = 150

FIGURE 5.6 – Taux d'acceptation des requêtes de type **unicast** en fonction de leur taux d'arrivée et des paramètres de simulation

5.4.4.1.1 Requêtes de type unicast Sur les graphiques 5.6a à 5.6d, le path-splitting montre un intérêt (de 2 à 3% de gain en fonction du taux d'arrivée) quelques soient les instances considérées. Les instances possédant un ratio de 10% montrent la plupart du temps de meilleurs résultats que les autres.

En revanche, sur ces mêmes graphiques, le paramètre β ne semble avoir quasiment aucun impact sur les résultats, puisque toutes les courbes de la figure 5.6 indiquent des taux d'acceptation semblables.



(a) $\beta = 1$

(b) $\beta = 50$

(c) $\beta = 100$

(d) $\beta = 150$

FIGURE 5.7 – Taux d'acceptation des requêtes de type **multicast** en fonction de leur taux d'arrivée et des paramètres de simulation.

5.4.4.1.2 Requêtes de type multicast Contrairement aux instances de type unicast, le path-splitting ne montre pas toujours un intérêt constant pour les instances de type multicast. Pour des taux d'arrivées faibles ($\lambda \leq 6$), ne pas utiliser le path-splitting semble le plus intéressant, alors que pour des taux supérieurs à 7 requêtes toutes les 100 unités de temps, l'utilisation du path-splitting avec un ratio de 10 ou 33% semble la meilleure solution.

Tout comme pour les instances de type unicast, le paramètre β ne semble pas avoir d'influence sur les résultats des simulations.

De manière générale, pour les instances de type multicast, les différences de taux d'acceptation sont plus faibles (1 à 2%) que pour les instances de type unicast (2 à 3%). Ceci s'explique par la quantité plus importante de bande passante demandée par les requêtes de type unicast ($\geq 7Gbps$). Il est plus probable que l'utilisation du path-splitting soit nécessaire pour un lien virtuel de grande capacité que pour un de faible

capacité.

5.4.4.2 Revenus totaux des simulations

Les figures 5.8 et 5.9 présentent les courbes de revenu des simulations. Le revenu est exprimé en **Mb** en supposant qu'une unité de temps soit égale à une seconde (pour des questions de simplicité dans les comparaisons). Dans la réalité, une unité de temps serait en général assimilée à plusieurs secondes (dans le cas d'un fournisseur de réseau, on peut supposer que les requêtes de réseau arrive à des taux inférieur à 1 requêtes toutes les 25 secondes, puisque le temps de déploiement est supérieur à la dizaine de minutes), ce qui impliquerait des différences entre les courbes plus importantes (deux fois plus importantes si une unité de temps correspondait à deux secondes, etc.).

5.4.4.2.1 Requêtes de type unicast Les courbes 5.8a à 5.8d montrent le revenu totale de différentes instances en fonction du taux d'arrivée de celles-ci. Elles confirment les résultats obtenus par les courbes montrant le taux d'acceptation : les instances utilisant le path-splitting montrent un revenu plus élevé que celles ne l'utilisant pas. La différence de revenu est d'autant plus importante que le taux d'arrivée des requêtes est élevé (sauf pour $\beta = 100$) : de 60 Tb pour $\lambda = 0.04$ à 200 Tb pour $\lambda = 0.1$.

Comme pour le taux d'acceptation, l'influence du paramètre β sur les revenus des simulations semble faible.

5.4.4.2.2 Requêtes de type multicast Les courbes de revenus des instances de type multicast (5.9a à 5.9d) sont beaucoup moins régulières que les courbes des instances de type unicast. Pour des taux d'arrivée faibles ($\lambda \leq 0.06$), les meilleurs instances sont celles n'utilisant pas le path-splitting avec $\beta = 1$, pour des taux élevés ($\lambda \geq 0.08$), les plus hauts revenus sont en général obtenus avec path-splitting.

Malgré leurs irrégularités, les courbes de revenus de la figure 5.9, comme celles de la figure 5.8, montrent que même si les variations du taux d'acceptation sont faibles, la différence dans les revenus générés est conséquente.

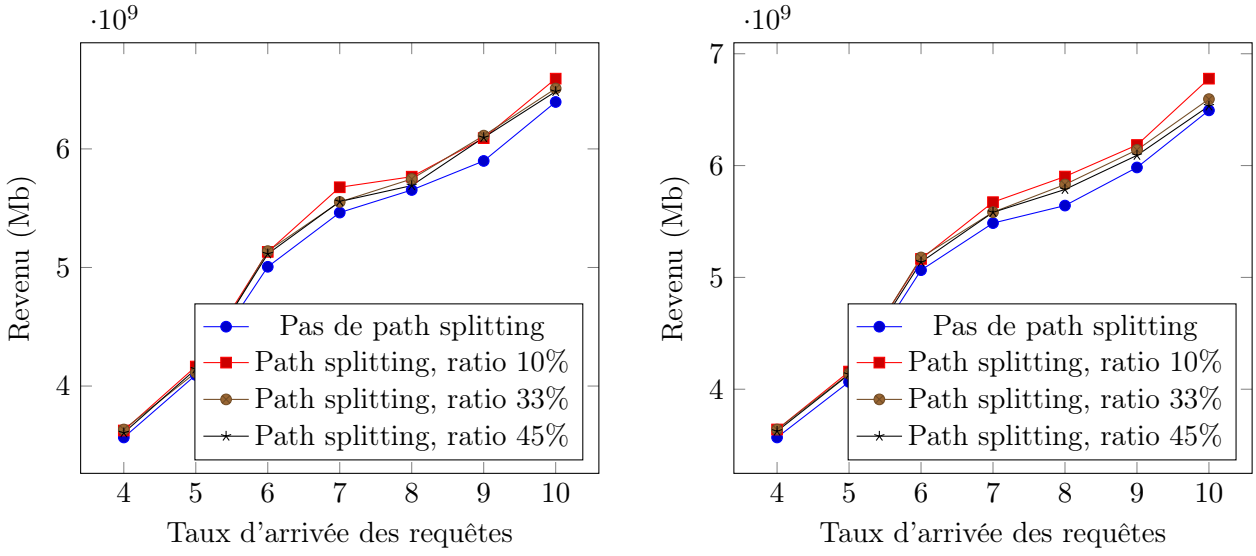
5.4.4.3 Charge du réseau

On ne considérera ici que des instances avec $\beta = 150$, les analyses précédentes ayant montré la faible influence de ce paramètre sur les résultats du modèle.

Les courbes 5.10a et 5.10b montrent la charge globale des liens du réseau (voir 5.3.3). Pour les instances de type unicast (graphique 5.10a), la charge du réseau dépend très peu des paramètres d'expérimentation, ce qui signifie qu'un meilleur taux d'acceptation des requêtes et de meilleurs revenus n'impliquent pas une surcharge notable du réseau.

Tout comme leurs courbes de revenus, les courbes des instances de type multicast 5.10b sont beaucoup moins régulières que celles des instances de type unicast. Pour un même type d'instance (caractérisé par β et PS_{ratio}), les courbes ne sont pas du tout linéaires en fonction du taux d'arrivée des requêtes.

Les différences entre les courbes des instances de type unicast et multicast, principalement au niveau des irrégularités, peuvent s'expliquer par une dépendance aux requêtes



(a) $\beta = 1$ (b) $\beta = 50$

(c) $\beta = 100$ (d) $\beta = 150$

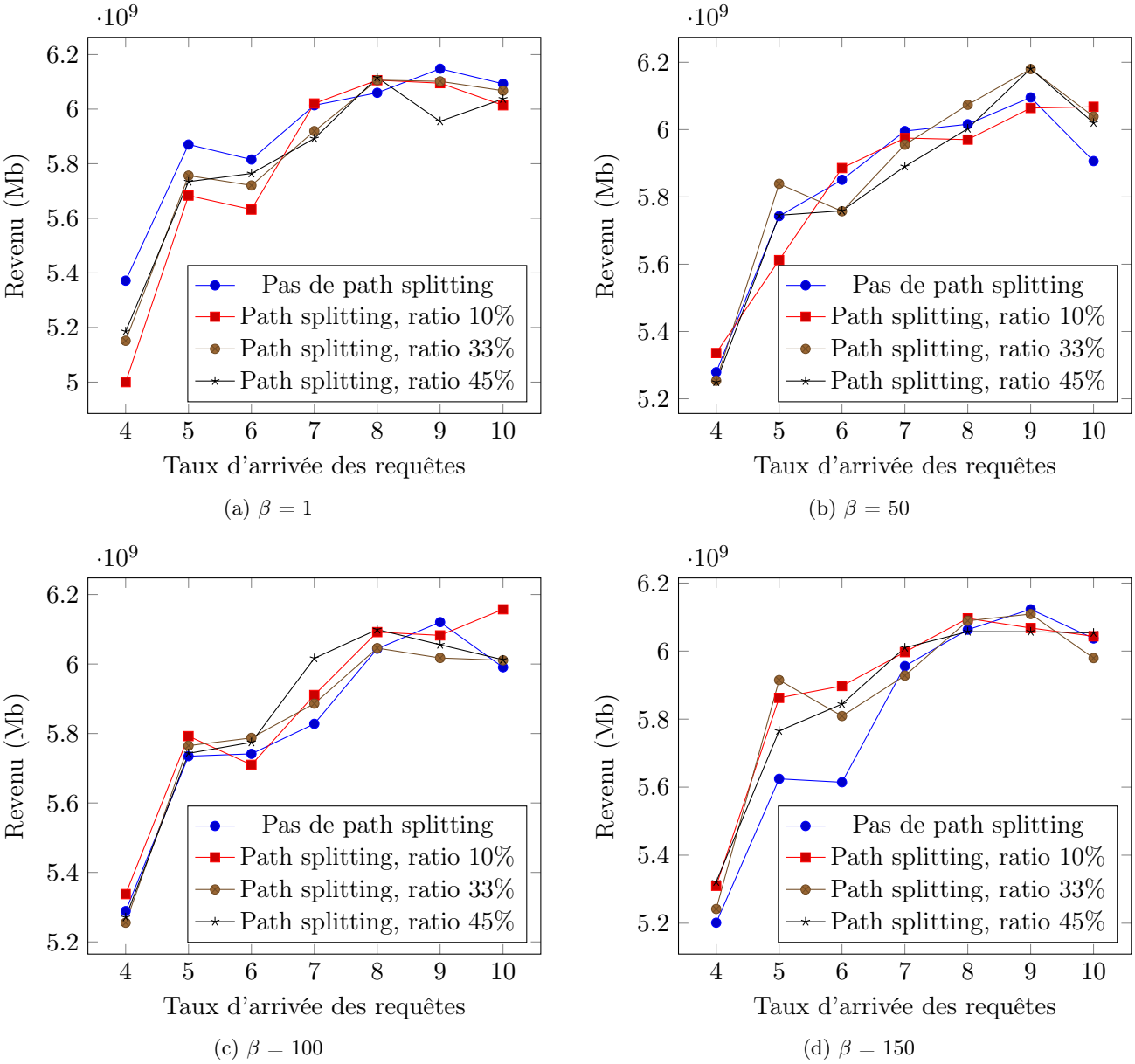
FIGURE 5.8 – Revenu totale de simulation pour des requêtes de type **unicast** en fonction de leur taux d'arrivée et des paramètres de simulation.

(nombre de liens, source, destination(s), etc.) beaucoup plus forte du côté des instances multicast.

5.4.5 Visualisation de l'équilibre de l'utilisation des liens dans le réseau

La figure 5.11 présente les fonctions de répartition du taux d'utilisation des liens du réseau (à la fin de la simulation, soit pour T = 10000 unités de temps) pour les instances de type unicast (5.11a) et multicast (5.11b) pour différents taux d'arrivée.

Comme sur le graphique 5.5d, on remarque que la fonction de répartition du taux d'utilisation pour des instances de type multicast varie peu en fonction du taux d'arrivée



(a) $\beta = 1$ (b) $\beta = 50$

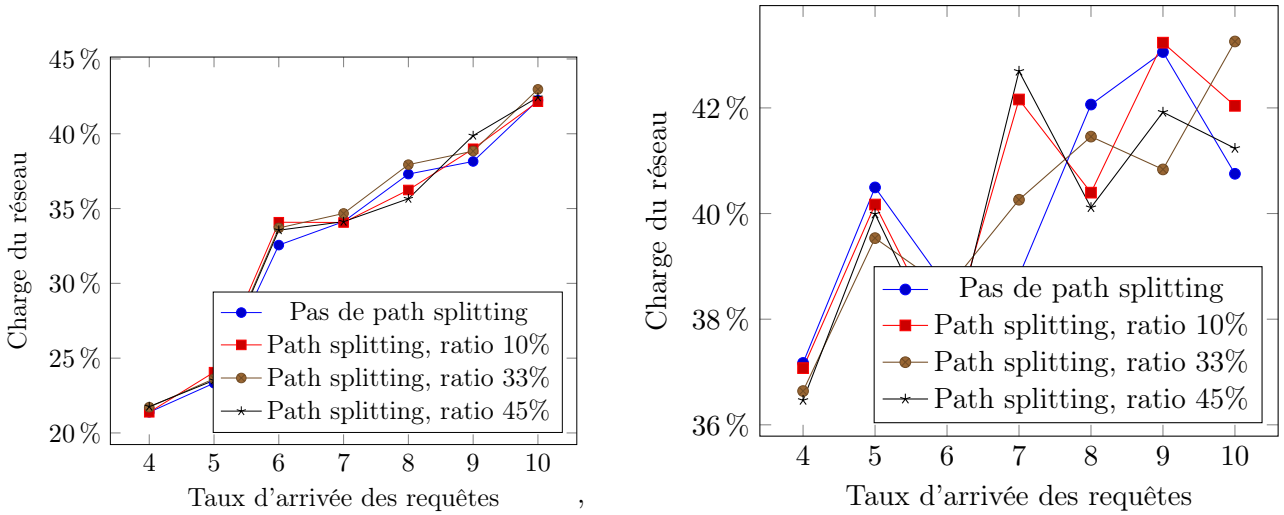
(c) $\beta = 100$ (d) $\beta = 150$

FIGURE 5.9 – Revenu totale de simulation pour des requêtes de type **multicast** en fonction de leur taux d'arrivée et des paramètres de simulation.

des requêtes (les courbes sont très proches sur le graphiques 5.11b). Quelque soit le taux d'arrivée des requêtes, environ 60% des liens sont utilisés à plus de 50% de leur capacité.

Le graphique 5.11a confirme les résultats des courbes 5.5c : le taux d'utilisation des liens du réseau augmente avec l'augmentation du taux d'arrivée des requêtes. Pour de faibles taux d'arrivées ($\lambda \leq 0.05$), moins de 20% des liens sont utilisés à plus de 50%, ce pourcentage atteint 50% pour un taux d'arrivée de 10 requêtes toutes les 100 unités de temps.

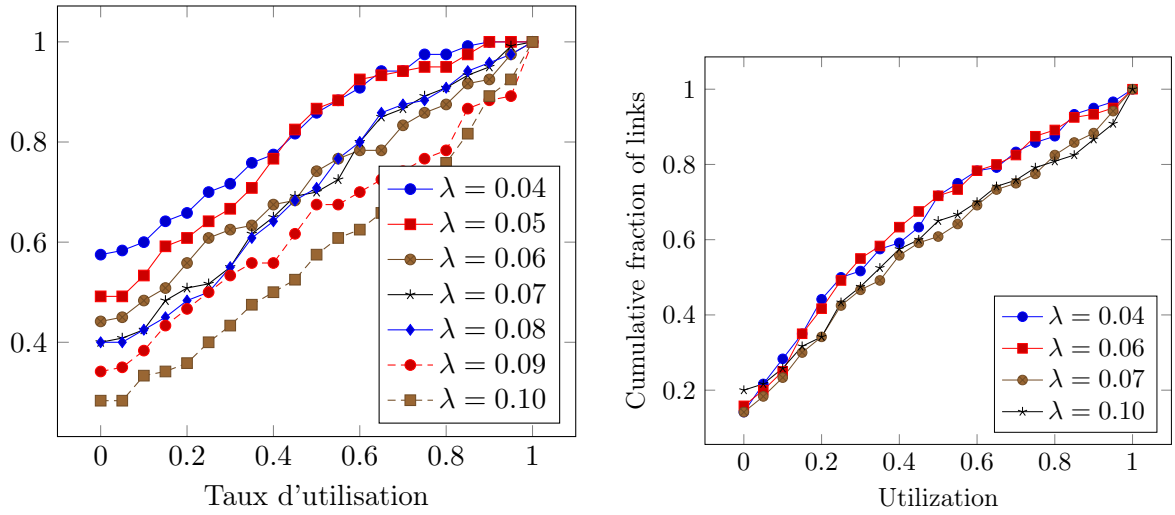
Quelque soit le taux d'arrivée, on remarque qu'au moins 20% des liens sont utilisés à plus de 80% de leurs capacités pour des instances de type unicast (60% pour celles de type multicast).



(a) Requêtes de type unicast

(b) Requêtes de type multicast

FIGURE 5.10 – Charge du réseau en fonction du taux d'arrivée pour des requêtes de type unicast et multicast - $\beta = 150$



(a) Requêtes de type unicast

(b) Requêtes de type multicast

FIGURE 5.11 – Fonction de répartition du taux d'utilisation des liens pour des requêtes unicast et multicast - $\beta = 150$, $PS_{ratio} = 10\%$

5.4.6 Analyse du taux d'acceptation des instances de type unicast et multicast

Il est important de noter que les seules requêtes de type unicast refusées par le modèle sont celles reliant deux points critiques du réseau (liens surchargés ou de faibles capacités). En supposant que le modèle essaie toutes les solutions possibles (ce qui semble être le cas puisque les temps limites de calcul fournit au solveur ne sont jamais atteints, voir 5.3), le refus d'une requête de type unicast reliant un routeur a à un routeur b implique qu'il n'existe **aucun** lien de capacité suffisante entre a et b.

En revanche, une requête contenant plusieurs liens de type multicast peut être refu-

sée à partir du moment où **un seul** de ses liens ne peut être alloué sur le réseau (une seule destination non atteignable). Dans les simulations effectuées, une requête de type multicast contient en moyenne 5 liens virtuels, chacun contenant en moyenne 5 destinations distinctes (ainsi qu'une source), soit 25 destinations pour un total de 30 routeurs. Si une seul de ces routeurs ne peut être « relié », alors la requête sera rejetée.

Le réseau GÉANT contient 41 routeurs, 16 d'entre eux étant reliés au cœur du réseau par des liens de capacités inférieures à 10 Gbps. Bien que le choix des cibles et destinations ne soient pas totalement aléatoire (voir 5.2.2.5), et qu'un routeur puisse être utilisé comme destination (ou source) de plusieurs liens au sein d'une même requête, la probabilité qu'un de ces routeurs isolés soit choisi n'est pas négligeable.

En supposant que tous les routeurs isolés soient connectés à un seul lien de 5 Gbps et que tous les autres routeurs soient reliés à deux liens de capacité 100 Gbps (cas pessimiste), la probabilité de choisir un routeur isolé serait 50 fois inférieure à celle de choisir un routeur « normal ». De façon simplifiée, on pourrait alors considérer que sur 50 routeurs tirés aléatoirement, un soit isolé, ce qui nous amènerait à avoir un routeur isolé toutes les deux requêtes.

En sachant que la capacité moyenne d'un lien virtuel multicast est de 1,4 Gbps, et en supposant le cas idéal où les routeurs isolés soient connectés par des liens de capacité 10 Gbps, sept requêtes seraient suffisantes pour saturer un de ces routeurs. En considérant que les routeurs isolés sont tirés les uns à la suite des autres (un toutes les deux requêtes), il faudrait $7 * 16 * 2 = 224$ **requêtes pour saturer l'ensemble des routeurs isolés** (sous l'hypothèse que les sessions restent actives suffisamment longtemps).

Ce chiffre de 224 requêtes peut être vu comme une borne supérieure du nombre réel de requêtes nécessaires à la saturation de tous les routeurs isolés, ce qui donnerait un temps de saturation d'environ 2240 unités de temps pour un taux d'arrivée $\lambda = 0.1$ et 5600 unités de temps pour un taux de 0.04. Une fois cette saturation atteinte, au minimum une requête sur deux sera refusée car contenant un de ces routeurs isolés.

5.5 Résultats expérimentaux sur le réseau de Campus

Les résultats obtenus avec le réseau de campus sont assez similaires à ceux obtenus avec le réseau Géant. Nous prsentons dans ce qui suit une synthèse.

5.5.1 Comparaison avec des méthodes heuristiques

La figure 5.12 présente le taux d'acceptation de notre algorithme et de l'heuristique la plus performante, i.e. PCC Coût $= g(c_e, \overline{c_e})$, en variant le taux d'arrivées des requêtes de type multicast. Comme pour le cas du réseau Géant, notre algorithme présente de meilleurs taux d'acceptation que l'heuristique considérée. Les différrences sont plus prononcées dans le cas du réseau de campus. En effet, avec l'heuristique, plusieurs chemins de données calculées ne respectent pas la contrainte de délai et sont donc rejetés (car le débit des liens est moins important que dans le cas de Géant et les exigences sur le délai maximum à respecter plus strictes).

La figure 5.13 présente les temps de convergence de notre algorithme et de l'heuristique en variant le taux d'arrivées des requêtes de type multicast. Les temps de convergence de notre algorithme sont certes supérieurs aux temps de convergence de

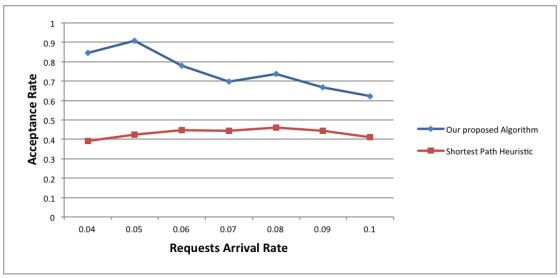


FIGURE 5.12 – Taux d'acceptation des reqêtes - Comparaison à l'algorithme PCC, Coût $= g(c_e, \overline{c_e})$

l'heuristique mais ils demeurent en moyenne de l'ordre de $60ms$.

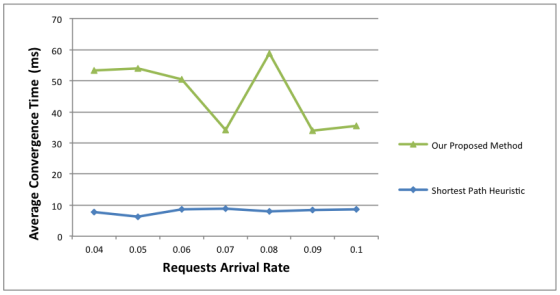


FIGURE 5.13 – Temps de convergence - Comparaison à l'algorithme PCC, Coût $= g(c_e, \overline{c_e})$

5.5.2 Caractérisation des temps de calcul

Les figures 5.14 et 5.15 décrivent les temps de convergence min, moyens et max en variant le taux d'arrivées des requêtes de type unicast et multicast. Ces temps demeurent inférieurs à $200ms$ (reps. $600ms$) pour les requêtes de type unicast (reps. multicast), ce qui est satisfaisant pour son application dans le contexte ADN.

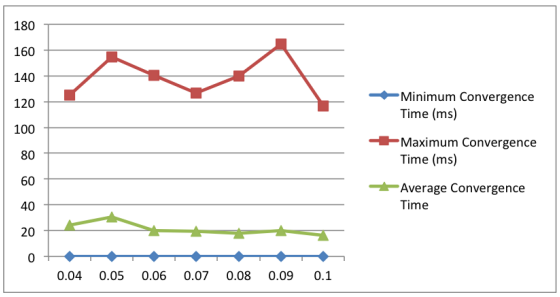


FIGURE 5.14 – Temps de convergence requêtes de type unicast

5.5.3 Taux d'utilisation des liens et des noeuds

Nous nous limitons ci-après aux résultats relatifs aux requêtes de type multicast. Les figures 5.16 et 5.17 présentent respectivement le taux d'utilisation des liens et des noeuds en variant le taux d'arrivées des requêtes. Les expérimentations montrent que certains

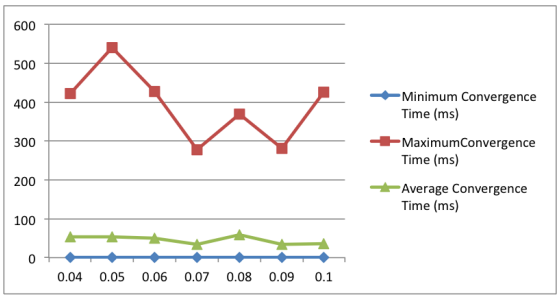


FIGURE 5.15 – Temps de convergence, requêtes de type multicast

liens (ceux de la dorsale du réseau) atteignent des taux d'utilisation supérieurs à 95%, ce qui explique le taux d'acceptation non nul. Le taux d'utilisation moyen est également élevé ce qui montre de nouveau l'efficacité de notre méthode. Pour les ressources de commutation, avec le modèle de charge considéré, les taux d'utilisation demeurent faibles, i.e. inférieurs à 20% (voir figure 5.17).

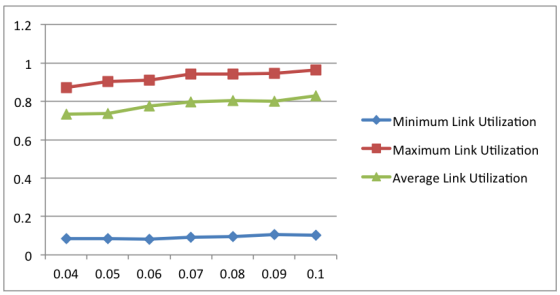


FIGURE 5.16 – Taux d'utilisation des liens, requêtes de type multicast

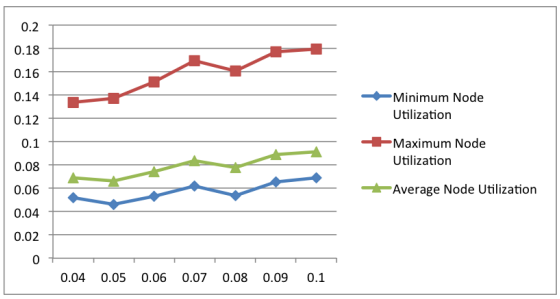


FIGURE 5.17 – Taux d'utilisation des noeuds, requêtes de type multicast

6. CONCLUSION

Ce document est le livrable de la tâche 2 (Cas d'application simulation distribuée: implémentation, validation et évaluation) du projet ADN dont l'objectif est (1) d'implémenter un prototype de l'approche ADN et de mettre en application l'approche pour offrir des services à une application de simulation distribuée dans un contexte réseau maîtrisé ; et (2) d'évaluer les algorithmes sur lesquels repose l'architecture ADN qui ont été introduit précédemment dans le livrable [1].

La première partie de ce rapport a présenté une mise à jour de l'architecture ADN dont le principal objectif est d'ouvrir l'application de l'approche ADN à des applications non nécessairement basées sur le middleware DDS. La principale nouveauté est le composant « Application Classifier » en charge de déceler la présence sur le réseau d'applications susceptibles d'utiliser le service ADN et d'en déduire ou évaluer leurs besoins.

La deuxième partie de ce document a décrit le prototype ADN qui a été implémenté. Les composants « Request Handler », « Resource Allocator », « VNET deployer » ont été entièrement développés ainsi qu'une instance du composant « Application Classifier » pour les applications DDS. Une version basique des composants « Flow Aggregator » et « Autonomic Manager » a été implémentée. Les algorithmes de ces composants sont amenés à être étendus et enrichis d'ici la fin du projet.

La troisième partie décrit le démonstrateur ADN qui a été développé pour illustrer la mise en application de l'approche ADN à une application de simulation distribuée impliquant plusieurs simulateurs de véhicule dirigés par des apprenants qui participent à un exercice ou une formation à la conduite en groupe. A travers un scénario applicatif simple, il illustre la détection, par le réseau ADN, de nouveaux besoins applicatifs puis le déploiement ou la mise à jour à la volée du service réseau correspondant. Il illustre également que le réseau ADN est capable de s'accommoder avec un niveau de granularité des besoins applicatifs très fin.

La quatrième partie a présenté certains résultats d'évaluation de l'algorithme d'allocation de ressources proposé sur la topologie réseau de Géant ainsi que sur la topologie réseau de campus de notre laboratoire. Les évaluations se sont focalisées sur des conditions aux limites (avec une forte surcharge), pessimistes par rapport à la réalité. Elles ont permis de vérifier la supériorité de notre algorithme par rapport à certaines heuristiques. Elles ont également permis de vérifier que les temps de convergence étaient acceptables, sauf pour quelques situations plutôt rares qu'il serait utile de mieux caractériser pour appliquer d'autres algorithmes basés sur des heuristiques qu'il serait utile de développer. Nos évaluations n'ont pas porté sur les autres algorithmes (« VNET Deployer » ou « Application Classifier ») car nous ne disposons pas d'une infrastructure réseau réelle suffisamment complexe pour nous permettre de mener des expérimentations pertinentes. Avec un micro-cloud en cours de déploiement au LAAS/CNRS, de nouvelles opportunités pour mener ces expérimentations pourraient voir le jour d'ici la fin de projet.

Dans la perspective du projet ADN, le travail présenté dans ce livrable servirait de base pour l'architecture finale et le prototype final du projet ainsi que pour le second démonstrateur prévu dans le cadre de la tâche 3.

6 REFERENCES

[1] S. Abdellatif, F. Simo Tegue, P. Berthou, T. Villemur « Livrable 1.2- Architecture préliminaire et algorithmes pour le cas d'applications dynamiques en environnement maîtrisé », Rapport de Contrat : Projet ANR- 13-ASTR-0024 ADN, Juillet 2015, 48p.

[2] S. Gorlatch, T. Humernbrum, and F. Glinka, "Improving QoS in real-time internet applications: from best-effort to Software-Defined Networks," in International Conference on Computing, Networking and Communications, 2014

[3] E. Egilmez, S. T. Dane, K. T. Bagci, and A. M. Tekalp, "OpenQoS: An OpenFlow controller design for multimedia delivery with end-to-end Quality of Service over Software-Defined Networks," in APSIPA ASC, 2012

[4] I. Bueno, J. I. Aznar, E. Escalona, J. Ferrer, and J. Antoni Garcia-Espin, "An opennaas based sdn framework for dynamic qos control," in IEEE SDN for Future Networks and Services, 2013

[5] P. Sharma, S. Banerjee, S. Tandel, R. Aguiar, R. Amorim, and D. Pinheiro, "Enhancing network management frameworks with SDN-like control," in IFIP/IEEE International Symposium on Integrated Network Management, 2013

[6] S. Tomovic, N. Prasad, and I. Radusinovic, "SDN control framework for QoS provisioning," in 22nd Telecommunications Forum Telfor, 2014

[7] M. F. Bari, S. R. Chowdhury, R. Ahmed, and R. Boutaba, "PolicyCop: an autonomic QoS policy enforcement framework for software defined networks," in IEEE SDN for Future Networks and Services, 2013

[8] T. Zinner, M. Jarschel, A. Blenk, F. Wamser, and W. Kellerer, "Dynamic application-aware resource management using Software-Defined Networking: implementation prospects and challenges," in IEEE Network Operations and Management Symposium, 2014

[9] A. Georgi, R. G. Budich, Y. Meeres, R. Sperber, and H. Hérenger, "An integrated SDN architecture for application driven networking," International Journal on Advances in Systems and Measurements, vol. 7, pp. 103–114, 2014.

[10] MRV, "Application-Aware Networking at A Glance." white paper, 2013

[11] Ben Alaya, M.; Monteil, T., "FRAMESELF: A Generic Context-Aware Autonomic Framework for Self-Management of Distributed Systems," in 21st IEEE WETICE, 2012

[12] Open Networking Foundation, "OpenFlow Switch Specification – version 1.4.0 (Wire Protocol 0x05)", Oct. 2013

[13] Object Management Group, "Data-Distribution Service for Real-Time Systems," OMG, version 1.4. Sept. 2014.

[14] A. Blenk, A. Basta, M. Reisslein, and W. Kellerer, "Survey on Network Virtualization Hypervisors for Software Defined Networking," 2015.

[15] S. Abdellatif, L. Bertaux, P. Berthou, S. Medjiah, A. Hakiri, T. Villemur, T. Gayraud, M. Diaz, T. Monteil, Livrable 1.1 « Etat de l'art : SDN, virtualisation réseau et DDS », Rapport de Contrat : Projet ANR- 13-ASTR-0024 ADN, octobre 2014, 74p

[16] ECA FAROS, Driving simulation, disponible sur : http://www.ecagroup.com/en/training-simulation/driving-simulation, consulté février 2016

[17] O. M. G. Specification, "The Real-time Publish-Subscribe Protocol (RTPS) DDS Interoperability Wire Protocol Specification v2.2," *Object Management Group pct/07-08-04*, Sep. 2014.

[18] Mininet : An Instant Virtual Network on your Laptop (or other PC), disponible sur: http://mininet.org, consulté mars 2016

[19] Pica8 P-3290 – 48 x 1GbE, Pica8 Inc., disponible sur : http://www.pica8.com/documents/pica8-datasheet-48x1gbe-p3290-p3295.pdf, consulté mars 2016
[20] CPqD, OFSoftSwitch 13. Disponible sur: http://cpqd.github.io/ofsoftswitch13/, consulté mars 2016
[21] Réseau Géant, Disponible sur:

ANNEXE A : DEFINITION DES TOPICS DE L'APPLICATION CONSIDEREE

```
module PlatSim
{
struct Position
{
long position;
};
struct caracteristique_vehiculeSim
{
long key;//@key
long systemeABS;
long rapportVolant;
long Roues;
long angleRouesStabLigne;
long vitRotationStabLigne;
short nbRouesMotrices;
short numRouesMotrices;
short numRouesDirectrices;
long moteur;
long boite;
long masseTotale;
long typeMoteur;
long carburant;
long couple_arbre;
long couple_essieu;
long cou_fort_emb;
long couple_resistant;
long CoeffDemarreur0;
long CoeffDemarreur1;
long carburant0;
long horloge0;
long embrayage;
long patinage;
long SurfaceAir;
short Cx;
    };

struct etats_vehiculesim
{
long key;//@key
long Volant;
long pVhlTracte;
long iFrequenceDynamique;
long iFrequenceDynamiqueVitesseReduite;
long vitessereduite;
long vAcceleration_visuel;
long amortisseur_angledevers;
long amortisseur_angle_roulis;
long momentInertie_a_la_roue;
short D;
short M;
long ag1;
long ag2;
long angle_remorque;
long cos_angle_remorque;
long sin_angle_remorque;
long vit_point_attelage;
long point_attelage_tracteur;
short point_attelage_remorque;
short distanceParcourue;
short pressionAir;
long temperFreinPied;
long temperFreinElec;
short volumeConsommee;
long consoInstantanee;
long bForceVhlSimul;
```

```
long moment_Arbre;
long  EcaDynaMoto_ModelState;
long EcaDynaVL_ModelState;
long couplevolant;
long coupleVolantPrec;
short niveauHuile;
short pressionHuile;
short temperEau;
long vitesseTurbo;
long coupleRoulements;
long couple_Frottements;
long couple_poids;
long couple_roue;
};

struct even_vehiculesim
{
long key;//@key
long accidentSec;
long noObjetColl;
long accident;
long essuieGlace;
long pannes;
long tabPassageSport;
long tabPassageConf;
long tabBlocage;
long modeCourantSelecteur;
long accelResulEssieu;
long chute;
long assistant_braquage;
long moteurMarche;
long demarreur;
long regulateur;
long couple_freinage;
long CoeffDerapL;
long CoeffDerapT;
long coupleMaxFreinMain;
long coupleEmbMax;
long boiteCraque;
long blocageRapoort;
};



struct VehiculeAutonome {
long key;//@key
long accelPrec;
long arml;
long armR;
long bDecalageForce;
long cercle;
long cercleConstruit;
long comportement1;
long comportement2;
long comportement3;
long comportement4;
long coupleMaxFreinRemorque;
long debutPhase;
long decalageSouhaite;
long depasseur;
long derniereBifurc;
long desequilibre_freinage;
long fDecalageForce;
long footL;
long footR;
long legL;
```

```
long legR;
long phase;
long phasePieton;
long pointsPasses;
long posVise;
long pVhlTracte;
long pVhlTracteur;
long rayonConstruction;
long stationneur;
long typePlacement;
};



struct exo
{long key;//@key
long exoConfigPermis;
long exoNbObjets3D;
long exoNbRoues3D;
long exoNomFichier;

};



struct ObjetStatique
{long key;//@key
long angleAxeX;
long angleAxeY;
long angleAxeZ;
long nAnimWayPoints;
long pAnimWayPoints;
long nAnimState;
```

```
long nAnimMode;
long fAnimSpeed;
long nAnimPos;
long pAnimFrameIRef;
long nAnimFrameDuration;
long nAnimFrameFirst;
long nAnimFrameLastPos;
long nAnimFramePosTD;
};


struct Objet3D
{long key;//@key
long Classe;
long Categorie;
long nomFicO3D;
long demiLarg;
long demiLong;
long hauteur;
long masse;
long section;
long idStructPos;
long idStructElt;
long idStructPoly;
long pere;
long nbSousObjets;
long nbLiens;
long liens;
boolean visible;
boolean actif;
long avertisseur;
long noSonavertisseur;
long noSonActivite;
```

```
boolean elimine;
float position;
float orientation;
float echelle;
long  sommets;
long rayonMax;
long collisionPoint;
};

struct FeuTricolore
{
long key;//@key
long etat;
long feux;
long nbPoints;
long noPoints;

};


struct PhotoScene

{
long key;//@key
long exitCode;
long CodeLangue;
long environnement;
long flags;
long horloge;
long dureeTrame;
long sensRoute;
char _reserved1[7];
char exoNomFichierVues[32];
```

```
char exoNomFichierTrafic[32];
long exoConfigPermis;
long exoNbObjets3D;
long exoNbRoues3D;
char _reserved2short[1];
char Reserved2[3];
boolean climatDistVisi;
long climatHeure;
long climatHorizon;
char reserved[3];
long obsPosition;
long obsRotation;
long obsNoObjVisu;
long obsTypeVue;
long obsReculX;
long obsReculY;
long obsReculZ;
char obsAnglesVues[6];
char obsAnglesVuesV[6];
long lMasqueVues;
char _reserved4[3];
long rainDensity;
long rainSize;
long wiperAngle;
long VitesseCode;
long LargeurVoie;
char Reserved5[3];
long VoitureVitArbre;
long VoitureCoeffDerapT;
long VoitureDerapL;
char VoitureSurfaceRoues[6];
long VoitureFlags;
long VoitureCle;
```

```
boolean VoitureEssuieGlace;
boolean VoitureFeux;
boolean VoitureClignotant;
long  VoitureCollision;
long VoitureDevers;
long VoitureRoulis;
boolean VoitureCarburant;
long VoitureDistanceParcourue;
boolean VoitureDecalageVoie;
long VoitureAnglesVoie;
long voitureVitesse;
long voitureDistVhlPrec;
long voitureConsoInstantanee;
char _reserved6[1];
char Userdata[32];
};



struct Climat
{long key;//@key
long climatDistVisi;
long climatHeure;
long climatHorizon;
boolean climatSport;
long rainDensity;
long rainSize;
long wiperAngle;

};
```

```
struct Photoroue3D
{long key;//@key
long noObjet;
long noRoue;
long Flags;
long Position;
long Rotation;
char _reserved0[1];
};


struct PhotoObjet3D

{long key;//@key
long noObjet;
long Classe;
long Flags;
long position;
long rotation;
long prop1;
long prop2;
long prop3;
long Infovoie;
char _reserved0[1];
};



struct Scene
{
long key;//@key
char nomScene[10];
long idTotal;
```

```
long idPartiel;
long idUnivTotal;
long idUnivPartiel;

};



struct Erreur
{long key;//@key
long idvehicules;
long position;
boolean criticite;
char message_erreur[100];
};



struct data_control
{ long key;//@key
    long idcontrol;
    char messagecontrol[100];

};
```