ETR 2026

12ème École d'Été Temps Réel

Actes de la session doctorants

07 – 11 septembre 2026

LIAS / ISAE-ENSMA

Poitiers / Futuroscope, France

https://etr2026.ensma.fr/

Comité de la session doctorants

Co-présidents

Yahya Hamdani
LIAS, ISAE-ENSMA

Merlin Kooshmanian
ONERA

Autres membres

Blandine Djika Mezatio
Lab-STICC, Université de Bretagne Occidentale

Sarah Dépernet
LORIA, Université de Lorraine

Alan Leboudec
Lab-STICC, Université de Bretagne Occidentale

Sebastien Levieux
Lab-STICC, Université de Bretagne Occidentale

Ouajih Mekni
LORIA, Université de Lorraine

Contact : `etr2026_phd@ensma.fr`

Sommaire

Une architecture pour fournir des communications sensibles au temps dynamiques en tant que service

Matthieu Amet
LORIA
Université de Lorraine, CNRS
F-54000 Nancy, France
matthieu.amet@loria.fr

***Résumé*—Les réseaux temps-réel sont largement utilisés dans des applications critiques pour la sécurité, tels que les systèmes avioniques ou encore les réseaux automobiles embarqués. La validation de leurs contraintes temporelles repose sur le calcul de bornes de latence garanties, généralement obtenues à l'aide de méthodes d'analyse déterministe comme le network-calculus.**

L'émergence de l'IETF Deterministic Networking vise à étendre ces garanties à des environnements interconnectant plusieurs réseaux temps-réels. Toutefois, les approches actuelles restent limitées à des systèmes administrés par une seule entité. En effet, le calcul des garanties de performance nécessite l'accès à l'ensemble des caractéristiques du réseau et des flux transportés, une hypothèse difficilement réalisable dans des environnements multi-domaines où les différents opérateurs ne se font pas mutuellement confiance.

Dans cet article, nous proposons une architecture permettant à des opérateurs de réseau de fournir des garanties de latence vérifiables pour les flux traversant leurs infrastructures grâce à l'utilisation de preuves à divulgation nulle de connaissance. Ces preuves certifient l'exécution correcte d'algorithmes de calcul réseau, notamment Total Flow Analysis (TFA) et son extension TFA++, tout en préservant la confidentialité des informations sensibles relatives aux flux et à la configuration du réseau. En complément, nous concevons un contrat intelligent chargé de gérer les engagements contractuels entre opérateurs, de coordonner les transactions financières associées et de vérifier automatiquement les preuves produites.

I. Introduction

Les réseaux temps-réel sont conçus pour fournir des garanties strictes sur les flux qu'ils transportent, notamment en termes de latence. Grâce à des méthodes d'analyse déterministe, il est possible de calculer des bornes supérieures de délai pour chaque flux et ainsi de vérifier le respect des contraintes temporelles imposées par les applications critiques.

Afin de satisfaire ces exigences, les équipements réseau peuvent mettre en oeuvre différents mécanismes de contrôle du trafic fournis par IEEE TSN, tels que l'attribution de priorités différenciées, le retardement de certains paquets ou encore leur suppression lorsque cela est nécessaire pour préserver les garanties offertes aux flux critiques. L'efficacité de ces mécanismes repose toutefois sur une connaissance précise des caractéristiques du réseau et des flux transportés. Cette nécessité de maîtriser l'ensemble du système explique pourquoi les réseaux temps-réel sont aujourd'hui principalement déployés dans des environnements fermés et fortement contrôlés, tels que les réseaux avioniques, industriels ou embarqués dans les véhicules autonomes.

L'intérêt croissant pour l'extension de ces garanties à des infrastructures de plus grande échelle a conduit à l'émergence de nouvelles initiatives de standardisation, en particulier du groupe de travail DetNet (Deterministic Networking) de l'IETF. L'objectif de DetNet est d'étendre les garanties déterministes offertes par les réseaux TSN au-delà des réseaux locaux traditionnels, afin de les rendre applicables à des infrastructures de plus grande envergure. Néanmoins, les architectures envisagées restent majoritairement limitées à des environnements administrés par une seule entité. Le groupe de travail précise d'ailleurs explicitement : ≪Nous ne chercherons pas à déployer des solutions pour des réseaux de la taille d'Internet ≫ [1]. Cette position reflète les difficultés inhérentes à la préservation de garanties déterministes dans des environnements composés de plusieurs domaines administratifs indépendants.

En effet, l'extension des réseaux temps-réel à des contextes multi-domaines soulève plusieurs défis majeurs. Le calcul des garanties de performance nécessite généralement l'accès à des informations détaillées sur la topologie du réseau, les mécanismes d'ordonnancement et les caractéristiques des flux. Or, dans un environnement impliquant plusieurs opérateurs indépendants, ces informations sont souvent considérées comme sensibles et ne peuvent être librement partagées. À cela s'ajoutent l'hétérogénéité des politiques d'administration, l'absence de mécanismes de coordination standardisés et le manque de garanties de confiance entre les différentes parties prenantes.

Ces limitations constituent aujourd'hui l'un des principaux obstacles à la généralisation des réseaux déterministes à grande échelle. Elles sont néanmoins remises en question par des travaux récents. En particulier, le draft de Bernardos et al. [2] propose un cadre générique pour l'interconnexion de plusieurs domaines DetNet. Cette approche traite les aspects liés au contrôle et à l'orchestration inter-domaines, mais laisse ouverte la question de

la confiance entre les entités administratives ainsi que celle de la vérification des garanties annoncées par chaque domaine.

Ainsi, malgré la maturité croissante des mécanismes permettant de garantir des propriétés temps-réel au sein d'un domaine unique, leur extension à des infrastructures multi-domaines demeure un problème largement ouvert. La mise en place de mécanismes permettant de préserver la confidentialité des informations internes tout en assurant la vérifiabilité des garanties offertes apparaît aujourd'hui comme une condition essentielle à l'émergence de réseaux déterministes multi-opérateurs.

Nous cherchons donc à combler cette lacune en permettant à un domaine réseau de démontrer cryptographiquement à une application critique cliente qu'il est en mesure de satisfaire les garanties temporelles associées au service demandé. Dans cette optique, nous introduisons le concept de DetNet as a Service (DNaaS) [3], dans lequel un opérateur de domaine DetNet, appelé DetNet Service Provider (DSP), agit comme fournisseur de services déterministes et fournit à ses clients des preuves vérifiables des garanties qu'il annonce.

À cette fin, nous exploitons les preuves à divulgation nulle de connaissance (Zero-Knowledge Proofs, ZKPs) afin de certifier l'exécution correcte d'un algorithme de calcul de bornes de latence. Un domaine peut ainsi démontrer que les garanties temporelles qu'il annonce ont été obtenues à partir d'un calcul valide, sans révéler les informations sensibles ayant servi à ce calcul, telles que la configuration du réseau, les chemins empruntés par les flux ou les paramètres d'ordonnancement. Cette approche établit un mécanisme de confiance vérifiable entre clients et fournisseurs de services DetNet, tout en préservant la confidentialité des informations internes des domaines administratifs.

II. Modèle du système

Dans cette section, nous décrivons le modèle du système. Celui-ci comprend deux types de participants : un opérateur, qui fournit des services de réseau déterministe, et des clients, qui sollicitent ces services pour leurs flux. Nous supposons que chaque participant dispose d'un accès à une blockchain publique, telle qu'Ethereum.

Nous présentons tout d'abord le modèle de l'opérateur et de son réseau, puis le modèle des accords de service et de configuration du réseau.

A. L'Opérateur et Son Réseau

Nous considérons un opérateur unique possédant et administrant un réseau à commutation de paquets capable de fournir des services déterministes. Ce réseau est composé de commutateurs TSN et/ou de routeurs DetNet interconnectés par des liens bidirectionnels complets (représentés en bleu dans la Figure 1).

Chaque commutateur ou routeur (représenté par une boîte dans la Figure 1) comprend plusieurs ports d'entrée,

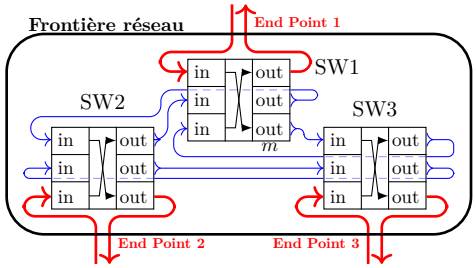


Figure 1 : Modèle du réseau de l'opérateur et ses *endpoints*.

plusieurs ports de sortie ainsi qu'une switching fabric. Nous considérons des équipements fonctionnant selon le principe du *store-and-forward*. Afin de nous concentrer sur notre solution, nous supposons que les ports de sortie utilisent un ordonnanceur à priorités strictes non préemptives comportant huit classes de trafic, la classe 7 étant la plus prioritaire et la classe 0 la moins prioritaire [**?**, §8.6.8.1]. Néanmoins, notre approche reste applicable à tout ordonnanceur TSN pour lequel des modèles fondés sur le calcul réseau sont disponibles.

Nous notons par $\mathcal{N}$ l'ensemble des ports de sortie du réseau. Pour un port de sortie $n \in \mathcal{N}$, nous notons B_n la taille de son tampon, c_n la capacité de transmission du lien sortant associé, supposée constante, et T_n une borne supérieure de la somme des latences technologiques au niveau de n et du délai de propagation du lien de transmission correspondant. Nous notons également s_n le commutateur contenant le port n et d_n le commutateur connecté à ce port par le lien de transmission. Par exemple, pour le port de sortie m de la Figure 1, nous avons $s_m = \text{SW1}$ et $d_m = \text{SW3}$.

La topologie physique du réseau de l'opérateur est supposée publique. Plus précisément, la topologie, décrite par l'ensemble des couples $(s_n, d_n)_{n\in\mathcal{N}}$, est communiquée à tous les clients, ainsi que les valeurs de c_n, T_n et B_n pour tout $n \in \mathcal{N}$. Nous supposons également que cette topologie est statique. Les clients peuvent ainsi s'appuyer sur des inspections ponctuelles sur site et avoir confiance dans le fait que la topologie annoncée par l'opérateur reflète fidèlement le réseau effectivement déployé.

Certains liens (représentés en rouge épais dans la Figure 1) traversent la frontière du réseau de l'opérateur au niveau de points de terminaison (*endpoints*). À ces points de terminaison, nous supposons que l'opérateur peut configurer des mécanismes de contrôle de trafic (*traffic policers*) [4, §8.6.5.5] afin de contrôler l'admission des flux et de rejeter les paquets excédant le profil de trafic qui leur est associé.

B. Accord de Service Déterministe

À l'instar d'un contrat de niveau de service (Service-Level Agreement, SLA), nous appelons accord de service déterministe (Deterministic-Service Agreement, DSA),

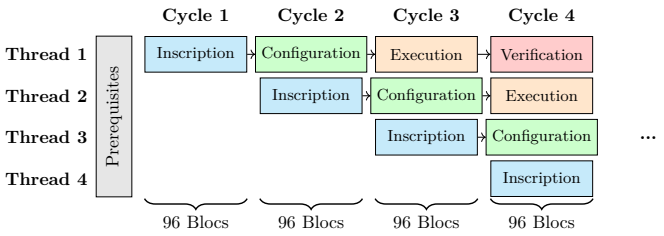


FIGURE 2 : Quatre cycles de vie DSA exécutés en parallèle, chacun progressant à travers ses différentes phases opérationnelles selon un cycle distinct.

noté $\mathcal{A}_f$, l'accord conclu entre un client et l'opérateur pour un flux DetNet donné f. Le contenu d'un DSA s'appuie sur les spécifications définies dans [5, §5] et [4, §34.6.1].

Le DSA décrit tout d'abord les caractéristiques du flux que le client s'engage à respecter au point d'entrée du réseau. Nous retenons l'interprétation dite ≪fenêtre glissante ≫(sliding interval) de cette spécification [6, §III.E] : sur tout intervalle de durée I_f, le flux peut émettre au plus $N_f^{\max}$ paquets, chacun de taille maximale $L_f^{\max}$. Le DSA précise également le point d'entrée $\mathrm{In}(f)$ ainsi que les points de sortie $\mathrm{Egs}(f)$ du flux. Enfin, il spécifie la date de début du service, sa durée ainsi que le montant versé par le client à l'opérateur en contrepartie de la fourniture du service.

Réciproquement, le DSA définit les garanties de service auxquelles l'opérateur s'engage. Dans ce travail, nous nous concentrons sur les garanties de latence bornée et d'absence de pertes dues à la congestion. Pour chaque point de sortie e, le DSA spécifie une latence maximale $\Theta_{f,e}$ que l'opérateur garantit entre le point d'entrée du flux et le point de sortie considéré.

Pour solliciter un service déterministe, le client transmet à l'opérateur une requête de DSA contenant l'ensemble des informations précédemment décrites. Le contenu de cette requête, à l'exception du montant associé au service, est considéré comme confidentiel et n'est accessible qu'au client concerné et à l'opérateur.

L'opérateur est libre de sélectionner les requêtes de DSA qu'il accepte de prendre en charge. Pour chaque DSA retenu, il attribue au flux f une `configuration` composée d'un chemin Φ_f reliant le point d'entrée aux différents points de sortie, ainsi qu'un niveau de priorité $p_f \in [0, 7]$. Cette attribution est supposée rester inchangée pendant toute la durée de validité du DSA.

III. ARCHITECTURE DNAAS

La figure 2 présente le déroulement complet des phases par lesquelles passe un DSA lorsqu'une requête est soumise à l'opérateur. Nous distinguons quatre acteurs principaux au sein de notre framework. Le client soumet des requêtes de DSA à l'opérateur. L'opérateur, propriétaire du réseau, décide d'accepter ou de rejeter ces requêtes, configure les chemins et les priorités des flux, puis génère les preuves ZKPs. Le contrat intelligent, déployé sur une blockchain compatible, vérifie les preuves à l'aide du vérificateur RISC Zero [7] afin de valider l'exécution correcte de l'algorithme de calcul réseau, puis orchestre les paiements entre les parties en fonction du résultat de cette vérification.

Les échanges entre ces acteurs sont organisés en quatre phases, chacune s'exécutant arbitrairement sur un cycle de 96 blocs (≈24 minutes). Ces phases s'enchaînent en pipeline, comme illustré en figure 2, permettant un service continu sans interruption entre les cycles.

A. Prérequis

Avant tout traitement de requête DSA, l'opérateur et les clients effectuent des étapes préliminaires. L'opérateur calcule un hachage cryptographique de la topologie réseau $\mathcal{H}(\mathcal{N})$, couvrant l'ensemble des ports de sortie $(s_n, d_n, B_n, T_n, c_n)_{n\in\mathcal{N}}$, le publie sur la blockchain via `init(`$\mathcal{H}(\mathcal{N})$`)` et partage la topologie $\mathcal{N}$ en clair avec les clients. Ce hachage servira ultérieurement à vérifier que la topologie utilisée lors de la génération de la preuve correspond bien à celle communiquée aux clients. Chaque participant dépose ensuite un collatéral sur le contrat intelligent via `addCollateral`, qui constitue une garantie financière pour le bon déroulement des paiements.

B. Inscription

Durant la phase d'inscription, les clients soumettent leurs requêtes DSA pour le cycle suivant. Pour chaque flux f, le client publie sur la blockchain le hachage $\mathcal{H}(\mathcal{A}_f)$ et le frais de service proposé via `addDsaRequest(`$\mathcal{H}(\mathcal{A}_f)$`,` `serviceFee`$_f$`)`, dont le montant est immédiatement verrouillé par le contrat. Simultanément, il transmet les données confidentielles du DSA directement à l'opérateur via un canal sécurisé. Cette double soumission lie les données publiques aux données confidentielles de façon vérifiable, sans les révéler sur la blockchain.

C. Configuration

La phase de configuration est entièrement conduite par l'opérateur, qui dispose d'une fenêtre de 80 blocs (≈20 minutes). Il commence par récupérer la liste des requêtes publiées sur la blockchain et les apparier avec les données confidentielles reçues via le canal sécurisé durant la phase d'inscription. Il sélectionne ensuite les DSA à admettre selon sa politique interne, et détermine pour chaque flux f de l'ensemble retenu $\mathcal{F}$ une configuration : un chemin Φ_f et une priorité $p_f \in [0..7]$ satisfaisant les exigences du DSA.

Une fois la configuration finalisée, l'opérateur exécute un algorithme de calcul réseau dans une machine virtuelle fournie par RISC Zero permettant le calcul de ZKPs. La figure 3 illustre le fonctionnement interne de la zkVM RISC Zero lors de la génération de la preuve. L'algorithme de calcul réseau (TFA ou TFA++ dans notre cas), écrit en Rust et partagé publiquement avec les clients, est compilé

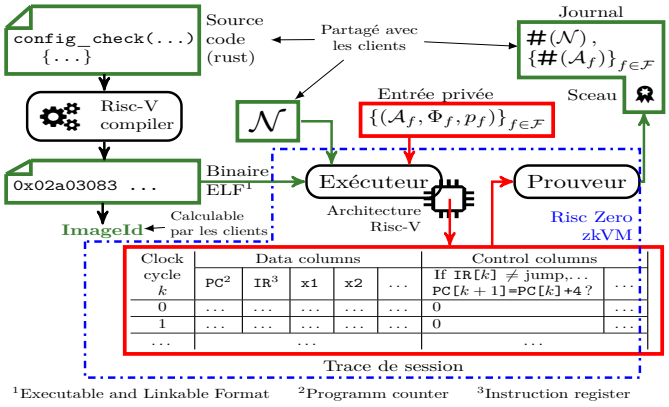


FIGURE 3 : Principe de l'exécution d'un code dans la machine virtuelle Risc Zero.

en un binaire au format ELF[1] ciblant l'architecture RISC-V. Ce binaire est identifié par une empreinte cryptographique, l'`ImageId`, que tout participant peut calculer de manière indépendante afin de s'assurer que la preuve porte bien sur l'algorithme attendu.

L'opérateur exécute ce binaire dans l'environnement RISC Zero en lui fournissant deux catégories d'entrées : les entrées publiques, à savoir la topologie réseau $\mathcal{N}$, et les entrées privées, à savoir l'ensemble des DSA sélectionnés avec leurs chemins et priorités assignés $\{(\mathcal{A}_f, \Phi_f, p_f)\}_{f\in\mathcal{F}}$. Durant l'exécution, l'exécuteur RISC-V enregistre l'intégralité des états mémoire et registres dans une *trace de session*, qui constitue le témoin de l'exécution. Le prouveur exploite cette trace pour générer deux artefacts. D'une part, le journal, qui contient les sorties publiques de l'algorithme : le hachage de la topologie $\mathcal{H}(\mathcal{N})$ et les hachages des DSA admis $\{\mathcal{H}(\mathcal{A}_f)\}_{f\in\mathcal{F}}$. Cet ensemble est partagé avec les clients. D'autre part, le sceau, qui constitue la preuve cryptographique que la trace de session résulte bien de l'exécution correcte du binaire identifié par l'`ImageId` sur l'architecture RISC-V, sans révéler les entrées privées.

L'opérateur soumet la preuve au contrat via `verifyAndPublishProof(sceau, journal)`. Le contrat délègue la vérification du sceau au vérificateur RISC Zero ; en cas de succès, il ancre le sceau on-chain, transfère les frais de service des DSA admis à l'opérateur, rembourse les clients des DSA rejetés, et publie le journal. Si l'opérateur ne soumet aucune preuve dans les 80 blocs impartis, les clients peuvent appeler `releaseDsaRequests()` durant les 16 blocs restants afin de récupérer leur collatéral.

D. Exécution & Verification

Les phases d'exécution et de vérification ne sont pas traitées dans ce papier et constituent des perspectives de travaux futurs. Nous en donnons néanmoins une description succincte afin de situer les contributions présentées dans le cycle de vie complet d'un DSA.

La phase d'exécution correspond à la période durant laquelle le réseau est effectivement configuré pour supporter les DSA admis. Sa durée est fixée à 96 blocs, débutant au premier bloc du cycle suivant la configuration. Des policeurs de trafic peuvent être déployés aux points d'entrée du réseau pour garantir que le trafic de chaque flux respecte le profil déclaré dans son DSA. Pour les services de longue durée, le client doit soumettre à nouveau sa requête DSA dans les cycles futurs.

La phase de vérification assure l'intégrité du comportement des clients et confirme la conformité de l'opérateur à la configuration annoncée. C'est cette phase qui permettrait, en définitive, l'orchestration finale des paiements, sous réserve qu'une configuration valide ait été prouvée lors de la phase de configuration et qu'elle ait été effectivement déployée lors de la phase d'exécution. Ces mesures post-exécution complètent la preuve formelle pré-exécution, offrant ainsi une couverture complète du cycle de vie du service.

IV. CONCLUSION

Nous proposons DNaaS, la première architecture combinant calcul réseau, ZKPs et contrats intelligents pour attester, auprès d'acteurs qui ne se font mutuellement pas confiance, de la capacité à fournir un service déterministe en contexte multi-tenant. Deux algorithmes de calcul de réseau, TFA et TFA++, ont été implémentés dans le framework RISC Zero afin de générer des preuves zero-knowledge de leur exécution correcte. Un contrat intelligent Solidity assure la vérification de ces preuves et l'orchestration des paiements sur une blockchain Ethereum.

Code source : https://gitlab.com/dnaas1/

RÉFÉRENCES

[1] "Deterministic Networking (detnet)," https://datatracker.ietf.org/wg/detnet/about/.

[2] C. J. Bernardos, L. M. Contreras, Q. Xiong, and A. Mourad, "A Control Plane Framework for Multi-Domain Deterministic Networking (DetNet)," Internet Engineering Task Force, Internet Draft draft-ietf-detnet-multi-domain-framework-00, May 2026, work in Progress.

[3] M. Amet, L. Thomas, and Y.-Q. Song, "DNaaS : Towards Scalable and Trusted Time-Sensitive Networks," in *IEEE INFOCOM 2026 - IEEE Conference on Computer Communications*, May 2026, pp. 1–10.

[4] "IEEE Standard for Local and Metropolitan Area Networks–Bridges and Bridged Networks," *IEEE Std 802.1Q-2022 (Revision of IEEE Std 802.1Q-2018)*, pp. 1–2163, Dec. 2022.

[5] B. Varga, J. Farkas, R. Cummings, Y. Jiang, and D. Fedyk, "Flow and Service Information Model for Deterministic Networking (DetNet)," Internet Engineering Task Force, Request for Comments RFC 9016, Mar. 2021.

[6] E. Mohammadpour, E. Stai, and J.-Y. Le Boudec, "Improved Network-Calculus Nodal Delay-Bounds in Time-Sensitive Networks," *IEEE/ACM Transactions on Networking*, vol. 31, no. 6, pp. 2902–2917, Dec. 2023.

[7] J. Bruestle, P. Gafni, and t. R. Z. Team, "RISC zero zkVM : Scalable, transparent arguments of RISC-V integrity," draft, August 11, 2023.

Ordonnancement et réflexion pour une prise en compte de la consommation énergétique

Maximilien Weinmann
Université de Namur et e-peas
Faculté d'Informatique
Namur et Louvain-La-Neuve, Belgique
Email : maximilien.weinmann@unamur.be

***Résumé*—Les ordonnanceurs ou schedulers présents dans des appareils IoT prennent rarement en compte l'énergie que consomment les différentes tâches qu'ils doivent ordonnancer. La plupart des appareils IoT sont des systèmes utilisant une ou plusieurs piles pour s'alimenter, et ne disposent d'aucun moyen de récupération d'énergie, fonctionnant comme si l'énergie des piles était éternellement abondante. Cela cause des problèmes de fonctionnement lorsque l'énergie vient à manquer mais aussi de maintenance du matériel en raison du remplacement des piles, et accroît l'impact environnemental lié à leur production régulière. Nous proposons une implémentation de système plus autonome, intégrant notamment un super-condensateur, un circuit de gestion de l'énergie, une source de récupération de l'énergie telle qu'une cellule solaire, ainsi qu'un scheduler tenant compte des dépenses et des apports énergétiques pour réguler l'exécution des taches. Les résultats expérimentaux sont actuellement en cours d'acquisition et feront l'objet de publications ultérieures.**

I. Introduction

Les ordonnanceurs sont utilisés dans les dispositifs de l'Internet des objets (IoT) depuis de nombreuses années, et depuis des décennies dans les ordinateurs. Ils servent à sélectionner, à tout moment, la tâche qui doit être exécutée par le processeur, en tenant compte de la priorité de la tâche et du temps nécessaire à son exécution. La littérature scientifique a largement étudié les ordonnanceurs, en expliquant leurs principes et l'impact de différentes techniques d'ordonnancement. Les chercheurs ont trouvé des moyens de minimiser le temps pendant lequel une tâche doit rester prête à s'exécuter, de réduire la latence afin qu'une autre tâche puisse commencer plus tôt, de maximiser le débit et d'augmenter la probabilité qu'une tâche reçoive la bonne quantité de temps d'exécution en fonction de sa priorité, de son temps d'exécution et de ceux des autres tâches.

La production de dispositifs IoT continue d'augmenter chaque année. Ces objets connectés servent à des usages variés dans de nombreux secteurs, notamment les systèmes de maison intelligente, les équipements de santé et de surveillance médicale, les systèmes de transport et automobiles, les technologies agricoles, les applications militaires, ainsi que dans bien d'autres domaines. Leur adoption est due à la facilité d'utilisation qu'ils apportent ; cependant, à mesure que leur nombre augmente, les problèmes associés, notamment ceux liés à la maintenance, augmentent également.

La miniaturisation des composants, comme la réduction de la taille des transistors, l'amélioration des architectures avec de meilleures conceptions de processeurs et des jeux d'instructions plus efficaces, ainsi que les systèmes de cache améliorés et les protocoles de communication innovants ont contribué au succès des dispositifs IoT. Cela les a rendus plus puissants sur le plan computationnel, donc plus rapides et plus intelligents, et par conséquent plus utiles. Cependant, comme tout dispositif, les objets IoT ont besoin d'énergie. Certains sont alimentés par le réseau électrique, ce qui les rend fiables, mais sans forcément tenir compte de leur consommation énergétique. D'autres sont alimentés par batterie, ce qui entraîne le remplacement régulier des batteries, augmentant les coûts de maintenance et l'impact écologique.

Les dispositifs IoT à récupération d'énergie constituent une solution à ce problème, car ils peuvent capter des énergies renouvelables, comme l'énergie solaire, soit pour une utilisation directe, soit pour stockage, et fonctionnent souvent avec des solutions de stockage alternatives et plus écologiques comme les super-condensateurs. Beaucoup d'entre eux nécessitent un ordonnancement, mais les ordonnanceurs ont principalement été conçus pour améliorer les performances de calcul. Dans ce document, nous analysons quelques méthodes d'ordonnancement récentes, et nous présentant le commencement de l'implémentation d'un nouvel ordonnanceur plus sensibles à l'énergie et mieux adaptées aux dispositifs IoT utilisant la récupération d'énergie ainsi que sa critique pour pouvoir l'améliorer et mieux répondre aux problèmes énoncés précédemment.

10 Juillet 2026

II. Œuvres connexes

Les dispositifs à récupération d'énergie, par opposition à ceux alimentés par le réseau électrique, présentent deux limitations majeures. Premièrement, ils ne peuvent stocker que de faibles quantités d'énergie qu'ils consomment rapidement. Deuxièmement, leur approvisionnement énergétique est imprévisible : il varie en fonction de plusieurs facteurs, notamment l'heure de la journée (en particulier pour les dispositifs alimentés par l'énergie solaire en extérieur), les conditions météorologiques et les déplacements éventuels du dispositif. Il est donc nécessaire de mettre en œuvre un mécanisme d'ordonnancement afin d'éviter les situations de pénurie d'énergie et d'assurer un fonctionnement continu du système.

Nous présenterons des solutions permettant à des dispositifs de fonctionner de manière intermittente. Ces solutions ont

été retenues en raison des innovations technologiques qu'elles apportent. Le terme solution est employé, car les technologies présentées regroupent un ensemble de composants matériels et une architecture logicielle, comprenant un ordonnanceur.

A. *Checkpointer*

Les approches fondées sur les checkpoints pour les systèmes fonctionnant de manière intermittente insèrent automatiquement ou manuellement des points de sauvegarde au cours de l'exécution du programme afin de préserver l'état du système et d'assurer la poursuite de son exécution malgré des coupures d'alimentation. Contrairement aux approches reposant sur des tâches délimitées explicitement, elles permettent de sauvegarder l'état du système, par exemple après une instruction ou un bloc de base, afin de reprendre l'exécution à partir du dernier état cohérent après une panne d'alimentation. Ces modèles s'apparentent ainsi davantage à un mécanisme de survie qu'à un véritable ordonnanceur temps réel.

Bien qu'ils offrent une plus grande flexibilité et nécessitent moins d'efforts de programmation que les approches basées sur les tâches, ils engendrent généralement un surcoût d'exécution plus important en raison des sauvegardes fréquentes de l'état et peuvent rencontrer des difficultés pour garantir la cohérence des données lors des coupures d'alimentation. Ils constituent néanmoins les premières solutions ayant permis de répondre au principal défi des dispositifs à récupération d'énergie : assurer un fonctionnement, même intermittent, malgré une alimentation électrique instable.
Clank est une solution basée sur le checkpointing, réalisée par Matthew Hicks [1]. L'objectif principal de Clank est d'assurer la poursuite correcte de l'exécution d'un programme malgré des coupures d'alimentation fréquentes et imprévisibles. Plusieurs approches viables permettent d'atteindre cet objectif. Pour guider ses choix de conception, le développeur de Clank propose un ensemble de principes que d'autres peuvent choisir ou non d'appliquer pour leurs propres solutions, tels que :

- Éviter de ralentir le matériel : Réduire la fréquence maximale du processeur a un impact direct sur la vitesse d'exécution du programme. Aussi, les bascules (flip-flops) non volatiles fonctionnent plus lentement que leurs équivalents volatils. Clank préserve la fréquence maximale d'origine du processeur. Il s'agit d'éviter une désynchronisation des données traitées par les différents composants matériels.
- Réduire le nombre d'instructions : Chaque instruction additionnelle nécessaire pour sauvegarder l'état réduit le nombre d'instructions disponibles pour faire progresser l'état du programme. Selon ses développeurs, Clank minimise cet impact, nécessitant moins de 1% d'instructions supplémentaires pour le checkpointing.
- Éviter d'augmenter la consommation d'énergie du matériel : L'énergie consacrée au matériel additionnel réduit l'énergie disponible pour l'avancement du programme. C'est l'inconvénient des méthodes qui nécessitent des modifications matérielles importantes ou qui dépendent de composants énergivores, comme les convertisseurs analogique-numérique utilisés pour surveiller les niveaux de tension. Il s'agit d'un principe que nous ne respecterons sûrement pas dû au besoin de calculer l'énergie récupérée et l'énergie restante du moyen de stockage.
- Éviter une ré-exécution longue : À mesure que les mécanismes de checkpointing s'améliorent, le surcoût lié à la ré-exécution peut devenir significatif.

Clank découpe dynamiquement les programmes qu'il doit exécuter en une série de séquences d'instructions redémarrables, afin de permettre la poursuite de l'exécution du programme en dépit des coupures d'alimentation. Il relie ces séquences à l'aide de points de contrôle, en veillant à ce que chaque séquence idempotente reste indépendante des accès mémoire des séquences précédentes. Le développeur de Clank introduit le concept d'ajout stratégique de points de contrôle afin d'équilibrer le surcoût de ré-exécution et celui du checkpointing. L'objectif idéal est d'éliminer les points de contrôle inutiles et d'en insérer uniquement lorsqu'ils sont nécessaires.

Pour détecter le code non idempotent, Clank utilise un "Read-first Buffer" contenant les adresses majoritairement lues, et un "Write-first Buffer" contenant les adresses majoritairement écrites. Clank vérifie ces tampons à chaque accès mémoire. Pendant l'exécution du programme, celui-ci tente d'accéder à certaines adresses. Si un accès à une adresse survient pour effectuer une écriture, mais que cette adresse se trouve dans le Read-First Buffer, une non-idempotence est détectée. Une détection de non-idempotence, selon des raisons précises que nous ne détaillerons pas ici, peut conduire à l'écriture d'un point de contrôle. Lorsque l'un des deux tampons est plein, Clank est contraint d'enregistrer un point de contrôle en mémoire non volatile.

Le système utilise également des "watchdog timers". L'un d'eux est le "Progress Watchdog". Son rôle principal est d'empêcher les programmes de rester bloqués à cause des coupures d'alimentation. Sans intervention, Clank pourrait créer des sections de code dont l'exécution prendrait plus de temps que la durée typique d'un cycle d'alimentation disponible (appelé "runt power cycle", ou cycle d'alimentation trop court). Cela entraînerait le système à tenter, encore et encore, d'exécuter ces longues sections, pour finir à chaque fois par perdre l'alimentation avant leur achèvement. Ces minuteurs optimisent les performances en trouvant le bon équilibre entre la fréquence de checkpointing et le surcoût de ré-exécution, tout en garantissant que les programmes continuent de progresser même en cas de coupures d'alimentation répétées. Néanmoins, cette solution, tout comme les autres solutions basées uniquement sur le checkpointing mentionnées précédemment, ne permet pas une prise en compte de l'énergie au niveau des tâches.

B. *Orienté tâche*

Les approches de programmation basées sur les tâches pour les systèmes intermittents divisent un programme en unités de calcul discrètes et atomiques appelées tâches. Chaque tâche doit être soit entièrement exécutée, soit ne pas être exécutée du tout ; les coupures d'alimentation ne sont autorisées qu'entre deux tâches. Ce modèle garantit la progression du calcul ainsi que la cohérence des données. En validant les résultats uniquement lorsqu'une tâche se termine correctement, il offre de fortes garanties d'atomicité tout en évitant le surcoût lié aux sauvegardes fréquentes de type checkpoint.

En revanche, il impose au programmeur de structurer explicitement son code en tâches et de gérer manuellement les dépendances entre celles-ci. Ce modèle se prête également plus naturellement à l'ordonnancement proprement dit, c'est-à-dire à la définition de l'ordre d'exécution des tâches afin d'optimiser l'utilisation du dispositif et de mieux gérer les interruptions liées aux pertes d'alimentation, bien que la possibilité de préemption des tâches doive être mise de coté. À l'inverse, les approches basées sur les checkpoints visent principalement à assurer la reprise après panne et la cohérence de l'exécution, sans traiter directement la planification des tâches. Les solutions fondées sur les tâches intègrent souvent des mécanismes de checkpointing, ce qui fait du checkpointing une composante partielle de la problématique du fonctionnement intermittent sur ordonnanceur.

L'ordonnanceur de tâches sensible à l'énergie conçu par Sabovic et al. [2], également appelé ordonnanceur EAS ou EAS, a été élaboré en prenant en compte l'intégralité du cycle de vie énergétique (c'est-à-dire le courant récolté, l'énergie stockée, ainsi que la consommation énergétique des tâches et de l'appareil), considérant cela comme le seul moyen d'éviter les coupures d'alimentation. Sur la base de cette perspective, les développeurs d'EAS ont reconnu la nécessité d'utiliser un outil de profilage afin de mesurer avec précision la consommation énergétique des tâches. EAS nécessite deux entrées : un binaire pré-compilé comprenant chaque tâche, ainsi que des métadonnées d'applications stockées au format JSON, ces métadonnées étant des paramètres de tâche tels que la deadline relative, le temps d'exécution, la consommation de courant et la priorité, le flux des tâches, c'est-à-dire leur ordre d'exécution, les contraintes des tâches, ainsi que le lien vers une tâche parente ou enfant. EAS met en œuvre un moyen d'éviter une répétition inutile de l'analyse du JSON : l'ordonnanceur vérifie d'abord si des données de tâches existent déjà en mémoire suite à une exécution précédente. Si des tâches y sont déjà stockées (ce qui peut se produire si le système a auparavant terminé la configuration initiale avant de subir une coupure d'alimentation), il charge ces instances de tâches existantes. Si aucune donnée existante n'est trouvée, l'ordonnanceur lit les définitions de tâches et les flux de travail à partir des fichiers de métadonnées JSON, les convertit en structures de données appropriées, puis les enregistre en mémoire non volatile. L'ordonnanceur EAS propose plusieurs méthodes d'ordonnancement : par assignation Rate Monotonic, par Earliest Deadline First, ou par une combinaison des deux, accompagnées de vérifications spécifiques liées à l'énergie et aux contraintes, comme le montre l'Algorithme 2 du même article [2]. Si EAS détecte un dépassement de deadline, qui peut également survenir en raison d'un temps de charge nécessaire, il peut soit supprimer l'instance de la tâche de la liste des tâches à exécuter, soit modifier sa priorité. C'est grâce au matériel adéquat, et en s'appuyant sur le modèle mathématique et les équations présentés dans un article précédemment publié par les mêmes auteurs [3], que l'ordonnanceur est capable de déterminer à quel moment suffisamment d'énergie a été récoltée pour exécuter une tâche. Ce modèle mathématique permet de faire, de manière optimale, le choix le plus fiable et le plus économe en énergie lorsque le niveau d'énergie est si faible qu'il impose de choisir entre éteindre l'appareil ou utiliser le mode veille. L'ordonnanceur de tâches est conçu pour optimiser l'utilisation de l'énergie et prévenir les coupures d'alimentation. Le système ne subira une coupure d'alimentation que lorsque l'énergie récoltée devient trop faible pour satisfaire ne serait-ce que les besoins minimaux en énergie de l'appareil en mode veille. EAS n'est pas l'ordonnanceur le moins consommateur, et son surcoût d'exécution n'est pas nécessairement le plus faible, puisque la consommation d'EAS pour exécuter une tâche dépasse celle d'InK (sachant qu'InK ajoute lui-même un surcoût à une tâche) d'environ dix pour cent, et un excédent similaire ou inférieur apparaît également concernant le temps d'exécution par rapport à InK. Cependant, l'ordonnanceur EAS garantit une meilleure prévention des coupures d'alimentation, c'est-à-dire une durée d'exécution de l'appareil plus longue lorsque la quantité d'énergie diminue, ce qui constitue une caractéristique considérable dans le monde de l'IoT à récupération d'énergie (EH IoT) et en matière de sensibilité énergétique.

III. Commencement d'un scheduler

Les morceaux de code présentés dans cette section sont destinés à être exécutés sur le MCU EDMS105N d'e-peas, basé sur une architecture ARM Cortex-M0. Le listing 1 présente la structure de données utilisée pour représenter une tâche, nommée task. Celle-ci contient un pointeur vers la fonction à exécuter, sa période d'exécution ainsi que l'instant de sa dernière exécution, exprimé en millisecondes. Cette structure est ensuite utilisée pour constituer une liste regroupant l'ensemble des tâches prises en charge par le scheduler. L'ajout d'une nouvelle tâche au système consiste simplement à ajouter une nouvelle entrée dans cette liste. Cette approche permet notamment d'intégrer progressivement différentes fonctionnalités, telles que la gestion d'un PMIC, d'un module Bluetooth ou encore les communications avec différents capteurs.

Dans la fonction main(), plusieurs éléments nécessaires au fonctionnement du système et au débogage sont initialisés. Un mécanisme de scheduling simple, naïf et non-energy aware est ensuite exécuté dans la boucle principale du programme, présentée dans le listing 2.

```
struct task {
    void (*run)(void);
    uint32_t interval_ms;
    uint32_t last_ms;
};

static struct task tasks[] = {
    {task_led, 500, 0},
    {task_sht31_sensor_communication,
       1000, 0}
};


int main(void)
{
    CMU_SetCoreClock(CMU, CMU_SRC_HFRO);
    CMU_EnablePeripheralClock(CMU, CMU_RTC
       );
    IORetarget_Init();
    RTC_EnableCounter(RTC);
    task_settings();
    uint32_t milliseconds = 0;
    uint32_t seconds = 0;
```

```
    while (1) {
        ...
    }
    return 0;
}
```

Listing 1. Structures et paramétrage dans main() pour le scheduler

Le scheduler implémenté ici repose sur une approche coopérative simple. À chaque itération de la boucle principale, le système parcourt successivement l'ensemble des tâches enregistrées et vérifie, pour chacune d'entre elles, si sa période d'exécution est arrivée à échéance. Cette décision est prise en comparant le temps courant avec l'instant de la dernière exécution enregistré dans la structure associée à la tâche. Lorsqu'une tâche doit être exécutée, son état est mis à jour puis sa fonction est appelée. Le scheduler n'est donc pas memoryless puisqu'il tient compte d'une certaine historicité d'exécution pour le choix de la prochaine tâche à exécuter.

```
while (1) {
    RTC_GetCounter(RTC, &seconds, &
        milliseconds);
    uint32_t now = seconds * 1000 +
        milliseconds;

    for (uint32_t i = 0; i < NUM_TASKS; i
        ++) {
        struct task *t = &tasks[i];
        if (now - t->last_ms >= t->
            interval_ms) {
            t->last_ms = now;
            t->run();
        }
    }
}
```

Listing 2. Intérieur de la boucle while du main()

Bien que cette approche soit adaptée à une première démonstration du fonctionnement d'un scheduler, elle présente plusieurs limitations. En particulier, son comportement dépend fortement de la durée d'exécution des tâches. Une tâche trop longue ou exécutée trop fréquemment peut monopoliser le processeur et empêcher les autres tâches d'être exécutées dans les délais attendus. Une amélioration possible consisterait à introduire un mécanisme d'interruption ainsi qu'une gestion du changement de contexte afin de permettre une meilleure isolation entre les tâches.

Dans le contexte des systèmes intermittents, la résilience aux coupures d'alimentation constitue également un enjeu majeur. Une méthode de checkpointing, similaire à celle proposée par Clank, pourrait être intégrée afin de sauvegarder l'état d'exécution du système et permettre une reprise après interruption. La structure task représentant une tâche pourrait également être enrichie avec des informations supplémentaires, telles qu'un niveau de priorité ou une deadline relative, afin de permettre des politiques de scheduling plus avancées et de profiter d'avantage du méchanisme d'interruption.

Afin d'exploiter pleinement les concepts liés à l'(EAS) de Sabovic, Delgado et Famaey, de nouvelles informations pourraient être associées à chaque tâche. Il peut s'agir de l'énergie disponible dans le système, de l'énergie nécessaire à l'exécution d'une tâche ou encore d'une estimation de l'énergie qui sera prochainement récoltée. L'intégration de ces paramètres permettrait de dépasser le cadre d'un scheduler coopératif classique pour évoluer vers un scheduler véritablement robuste et conscient de son environnement énergétique.

Une question centrale concerne alors le comportement à adopter selon l'état énergétique du système. Le scheduler doit-il favoriser l'exécution immédiate des tâches en situation d'abondance énergétique ou bien donner priorité à une recharge quitte à délaisser la notion de work-conservism ? Limiter les consommations en période d'équilibre ? Suspendre certaines activités lors d'un déficit énergétique ? Si oui, en fonction de quels critères ? Ces choix représentent une opportunité de différenciation par rapport aux approches de scheduling energy-aware existantes.

L'utilisation de prédictions énergétiques implique cependant une évolution du modèle de scheduler. La simple conservation de la dernière date d'exécution d'une tâche pourrait devenir moins pertinente, tandis que d'autres informations historiques pourraient être conservées afin d'améliorer les capacités de prédiction. Le scheduler ne serait donc pas nécessairement memoryless, même si certaines informations temporelles traditionnelles disparaissaient.

De la même manière, la propriété de work-conservation pourrait être remise en question. Dans un système alimenté par récupération d'énergie, maintenir l'appareil opérationnel peut être plus important que maximiser l'utilisation immédiate du processeur. Le scheduler pourrait alors introduire une distinction entre deux dimensions de priorité.

La première correspondrait à une priorité d'exécution traditionnelle, liée aux contraintes temporelles ou fonctionnelles des tâches. Les tâches possédant une priorité élevée pourraient ainsi préempter les tâches moins prioritaires, selon un mécanisme similaire à celui utilisé dans les politiques de scheduling temps réel telles que l'EDF (Earliest Deadline First).

La seconde dimension représenterait l'importance intrinsèque de l'exécution d'une tâche pour le maintien du fonctionnement global du système. Certaines tâches pourraient ainsi être considérées comme essentielles à la continuité opérationnelle de l'appareil, même si leur priorité temporelle est faible. En cas de déficit énergétique, ces tâches pourraient être favorisées au détriment de tâches plus prioritaires selon les critères classiques, afin d'éviter une extinction du système ou une dégradation critique de son fonctionnement.

Cette approche permettrait donc de dissocier deux notions habituellement confondues : la priorité liée aux contraintes d'exécution, notamment la fréquence ou les échéances des tâches, et la priorité liée à leur importance pour la survie et la continuité du système.

Avant de procéder à ces choix d'implémentation et aux différentes améliorations envisagées, le scheduler naïf présenté dans les listings 1 et 2 permettra d'établir des Figures of Merit. Ces indicateurs serviront de référence expérimentale afin d'évaluer les performances du système initial et de quantifier les améliorations apportées par l'intégration d'un scheduler Energy-Aware.

IV. Conclusion

Dans cet article, nous présentons quelques avancées récentes et importantes dans le domaine des systèmes intermittents, à savoir le checkpointing et le scheduling orienté tâche. Nous faisons part de notre plan pour implémenter un nouveau scheduler energy-aware profitant de ces avancées, et nous nous projetons sur quelques interrogations que le scheduler devra résoudre. Toute une série de mesures de consommation de tâches, de prédiction de récolte d'énergie, et de capacité du futur scheduler sont encore à venir et pourront être présentées dans un article à venir.

Acknowledgment

L'auteur aimerait remercier C. Hocquet, Team Leader e-peas, ainsi que L. Schumacher et P-Y. Schobbens pour leur guidance et soutien.

Références

[1] M. Hicks, "Clank : Architectural support for intermittent computation," in *2017 ACM/IEEE 44th Annual International Symposium on Computer Architecture (ISCA)*, 2017, pp. 228–240. [Online]. Available : https://ieeexplore.ieee.org/document/8192474

[2] A. Sabovic, A. K. Sultania, C. Delgado, L. D. Roeck, and J. Famaey, "An energy-aware task scheduler for energy-harvesting batteryless IoT devices," *IEEE internet of things journal. - Piscataway, NJ, 2014, currens*, vol. 9, no. 22, pp. 23 097–23 114, 2022, conference Name : IEEE Internet of Things Journal. [Online]. Available : https://ieeexplore.ieee.org/abstract/document/9803046

[3] A. Sabovic, C. Delgado, D. Subotic, B. Jooris, E. De Poorter, and J. Famaey, "Energy-aware sensing on battery-less LoRaWAN devices with energy harvesting," *Electronics 2020, 9, 904*, vol. 9, no. 6, p. 904, 2020, number : 6 Publisher : Multidisciplinary Digital Publishing Institute. [Online]. Available : https://www.mdpi.com/2079-9292/9/6/904

Formal Modeling and Schedulability Analysis of HPC-DAGs Using Time Petri Nets

Houda KHADIRI[1,2], Khaoula BOUKIR[2], Pierre-Emmanuel HLADIK[1], Houssam-Eddine ZAHAF[3]
[1]*Ecole Centrale de Nantes-LS2N-Nantes-France*
[2]*Université Ibn Tofail-setime-Kénitra-Morocco*
[3]*Nantes Université-LS2N-Nantes-France*
houda.khadiri@ec-nantes.fr, khaoula.boukir@uit.ac.ma
pierre-emmanuel.hladik@ec-nantes.fr, houssam-eddine.zahaf@univ-nantes.fr

***Abstract*—The HPC-DAG task model provides an expressive abstraction for heterogeneous real-time applications with alternative implementations and conditional execution paths. This expressiveness complicates schedulability analysis, especially under global scheduling, because both design-time choices and runtime behaviors must be considered. To address this challenge, we propose a Time Petri Net (TPN) formulation that models the timing behavior of HPC-DAG tasks executing on heterogeneous multiprocessor platforms and enables their formal schedulability analysis. Alternative implementation choices are encoded as controllable transitions and analyzed by controller synthesis in Romeo model-checker, while runtime choices remain uncontrollable. The objective is to determine whether a selection of alternatives exists such that no deadline-miss state is reachable. The proposed formulation constitutes a first Romeo-based verification workflow for HPC-DAG schedulability analysis.**

Keywords*—*Real-time systems, heterogeneous platforms, schedulability analysis, Time Petri Nets, HPC-DAG tasks, Romeo model-checker.

I. INTRODUCTION

Modern embedded real-time platforms increasingly combine general-purpose processors with specialized accelerators such as GPUs, DLAs, and copy engines. This heterogeneity improves computational capacity but complicates allocation and timing analysis, since a software component may have multiple implementations with different execution costs. Classical sporadic and periodic task models represent sequential workloads, while DAG (Directed Acyclic Graph)-based models represent parallel tasks as computation nodes connected by precedence constraints [1]–[3]. Conditional DAG models further represent runtime-dependent branches [4], but do not directly capture the combination of heterogeneous resources, alternative implementations, and conditional execution paths.

The HPC-DAG (Heterogeneous Parallel Conditional DAG) task model [6] addresses this combination by extending DAG tasks with alternative nodes for design-time implementation choices and conditional nodes for runtime-dependent branches. Thus, alternative selection determines one concrete implementation before execution, whereas conditional branches must be handled during execution. Existing HPC-DAG analyses mainly rely on analytical schedulability tests and allocation heuristics [6], [7], these approaches are scalable but may be pessimistic.

Time Petri Nets (TPNs) [8] provide a natural formalism for modeling this behavior: precedence constraints are represented by place/transition arcs, execution times by firing intervals, and concurrent branches by Petri-net concurrency. A TPN extends a classical Petri net by assigning a firing interval to each transition, constraining when it may fire. However, schedulability requires finding one valid selection of alternative implementations while accounting for all uncontrollable runtime behaviors. We therefore formulate schedulability as a controller-synthesis problem, where alternative choices are represented by controllable transitions and runtime behavior remains uncontrollable. The resulting model is verified using the Romeo model-checker [10].

The contributions of this paper are: (i) a TPN model for globally scheduled heterogeneous HPC-DAG task sets; and (ii) a Romeo-based verification formulation in which alternative choices are controllable decisions and deadline misses are forbidden states.

II. SYSTEM MODEL

A. Architecture Model

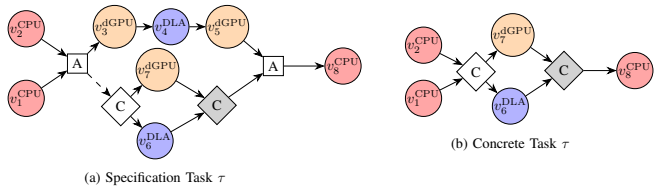
We consider a heterogeneous platform composed of a set of execution engines $E = \{e_1, e_2, \ldots, e_m\}$.

Each engine $e \in E$ is characterized by an engine tag $tag(e) \in T_{tag}$ describing its execution capabilities, for instance CPU, GPU, DLA, or copy engine, and by a local scheduling policy. A computation node can execute only on an engine with a compatible tag. In this short paper, we assume non-preemptive global fixed-priority scheduling within each class of engines sharing the same tag, although the modeling approach can be adapted to other policies.

B. HPC-DAG Task Model

1) Specification tasks: A specification task Fig. 1 is a DAG $\tau = (T, D, N, E_\tau)$, where T is the period, $D \leq T$ is the relative deadline, N is the set of nodes, and $E_\tau \subseteq N \times N$ is the set of precedence edges. The node set is partitioned as $N = V \cup A \cup \Gamma$, where V is the set of computation nodes, A is the set of alternative nodes, and Γ is the set of conditional nodes. Each computation node $v \in V$ represents an executable sub-task and is characterized by a tag $tag(v)$ and a worst-case execution time $C(v)$.

It may execute only on an engine e such that $tag(e) = tag(v)$. Alternative nodes $a \in A$ model design-time choices between



(a) Specification Task τ

(b) Concrete Task τ

Fig. 1. Specification and concrete HPC-DAG tasks.

several implementations of a node or subgraph. During offline analysis, exactly one outgoing branch is selected and the others are removed. This is the mechanism used to represent, for example, that the same functionality may be implemented on a CPU or on an accelerator with different execution costs.

Conditional nodes $\gamma \in \Gamma$ model runtime choices. Exactly one outgoing branch is taken during a job execution, but the chosen branch is not known before runtime and may vary from one job to another. This is the key semantic difference between A and Γ: alternatives are configuration decisions controlled during analysis, while conditions are uncontrollable runtime behaviors that must all be considered by verification.

For a node n, $pred(n)$ and $succ(n)$ denote the immediate predecessor and successor sets. An edge $(n_i, n_j) \in E_\tau$ enforces that n_j can start only after n_i completes. Source nodes are activated when a job is released; sink nodes must complete no later than the job deadline.

2) Concrete tasks: A concrete task Fig. 1 is an instance of a specification task in which all alternative choices have been resolved. Formally, a concrete task derived from τ is $\bar{\tau} = (T, D, \bar{V}, \bar{\Gamma}, \bar{E}_\tau)$. It contains no alternative nodes, since each $a \in A$ has been replaced by exactly one selected branch. Conditional nodes are preserved because they represent runtime behavior.

In this work, we consider a periodic task model. Each HPC-DAG task τ releases jobs separated by T time units. For a job released at time a, all source nodes are activated at a. A node becomes eligible for execution only after all its predecessors have completed. All sink nodes must complete within the interval $[a, a + D]$.

III. TIME PETRI NETS

We use extended Time Petri Nets (TPNs) with finite-domain variables, guards, and update functions. These extensions allow the model to represent dynamic node readiness, resource availability, branch selection, and scheduling decisions.

Definition 1. An extended Time Petri Net is a tuple $P = (Pl, Tr, X, Pre, Post, guard, update, M_0, x_0, I)$, where Pl is a finite set of places, Tr is a finite set of transitions, X is a finite set of bounded integer variables, Pre and $Post$ define the input and output arcs, $guard$ specifies the conditions enabling transitions, $update$ defines variable updates when transitions fire, M_0 and x_0 are the initial marking and variable valuation, and $I : Tr \to \mathcal{I}(\mathbb{Q}_{\geq 0})$ assigns a firing interval to each transition. A transition is enabled when its input places contain the required tokens and its guard is satisfied. If $I(t) = [a_t, b_t]$, transition t cannot fire before a_t time units after becoming enabled and must fire no later than b_t, unless it is disabled.

While model checking verifies whether a model satisfies a given property, controller synthesis computes a strategy that guarantees the property for all possible system executions. The strategy restricts the firing of controllable transitions, while uncontrollable transitions remain under the control of the environment. Hence, the synthesis problem consists in determining whether such a strategy exists and, if so, computing it.

To model this interaction, transitions are partitioned into controllable and uncontrollable sets, $T_r = T_c \cup T_u$. During execution, controllable transitions may be disabled, whereas enabled uncontrollable transitions cannot be prevented from firing. Therefore, the synthesized strategy must guarantee the target property regardless of the choices made by uncontrollable transitions.

A state space of P consists of its current marking and the elapsed time of each enabled transition. States sharing the same marking are grouped into a state class c, which records ranges of possible elapsed times. Let C denote the set of reachable state classes, $en(c)$ the transitions enabled in class c, and T_c the set of controllable transitions. A control strategy assigns to each reachable class $c \in C$ a set $\sigma(c)$ of controllable transitions allowed to fire, with $\sigma(c) \subseteq en(c) \cap T_c$. Thus, $\sigma(c)$ represents the control decision at class c: selected controllable transitions remain allowed, whereas unselected ones are disabled; uncontrollable transitions cannot be blocked. A strategy is winning for a property ϕ if ϕ holds for all behaviors consistent with these decisions. Romeo computes the most permissive winning strategy and reports success if one exists from the initial state.

IV. HPC-DAG SCHEDULABILITY ANALYSIS USING TIME PETRI NETS

The TPN construction follows the operational semantics of HPC-DAG jobs. Fig. 2 shows an elementary pattern with one alternative block and two branches. For readability, yellow rectangles denote transitions and blue circles denote places. For each computation node v, $v.ready$ and $v.exec$ represent the ready and execution states, while $v.start$ and $v.end$ denote the beginning and completion of its execution. The labels alt_0 and alt_1 identify the alternative branches, alt_node the decision point, and $join\_alt$ and $closing\_alt$ the synchronization and closing of the alternative block. The firing interval below each transition specifies its allowed firing time, while additional annotations represent guards and update operations used for node activation, branch selection, and scheduling. The branch transitions alt_0 and alt_1 are controllable: the Romeo controller may authorize one of them to select the implementation. The selected computation branch is then executed through timed completion transitions, after which the branches are joined and the task reaches its termination state.

A. Periodic Activation

For each task τ_i, a transition with interval $[T_i, T_i]$ releases successive jobs (cf. Figure 2-Phase A). Since the lower and upper bounds are equal, the interval represents a fixed delay: the transition fires exactly T_i time units after becoming enabled, thereby releasing a new job every T_i time units. The first job is enabled at time zero. At each release, source nodes become eligible, whereas non-source nodes remain inactive until their predecessor counters are satisfied.

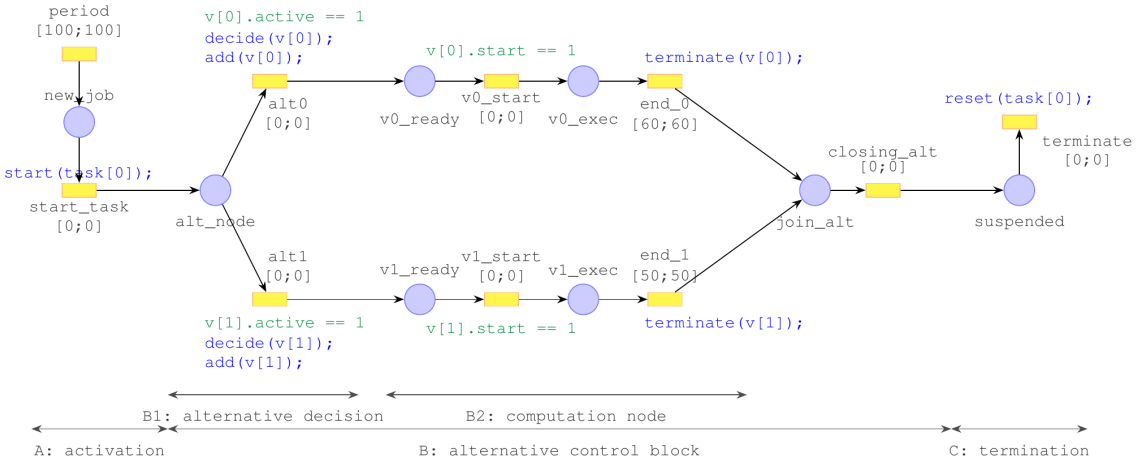


Fig. 2. Time Petri Net model of an HPC-DAG task with two alternative nodes.

B. Computation Nodes

A computation node v is represented by a ready place, a start transition, an execution place, and a completion transition. In Figure 2, this construction is illustrated for the two computation nodes v_0 and v_1, which belong to the alternative branches alt_0 and alt_1, respectively. A node cannot start execution before all its precedence constraints are satisfied, the start transition is enabled only when the scheduler assigns v to an eligible engine and the completion transition has interval $[C(v), C(v)]$ under WCET semantics, conservatively modeling execution at its worst-case duration; $[BCET(v), WCET(v)]$ can be used to model execution-time variability.

C. Alternative and Conditional Nodes

Alternative and conditional nodes are modeled using similar Petri-net branching structures. A control node is represented by as many transitions as outgoing branches, each transition corresponding to one possible branch.

For alternative nodes,all branches are initially enabled. Once a branch is selected, it remains active for the entire analysis, while the other alternatives are permanently disabled.

Each control structure is delimited by an opening control node and a corresponding closing node, as illustrated in Figure 2.

Alternative nodes introduce a fundamental difference with respect to classical Petri-net verification. in a standard TPN analysis, all reachable executions must be explored. In contrast, schedulability of an HPC-DAG task set requires finding at least one valid selection of alternative branches that avoids deadline violations. Thus, in our model, transitions associated with alternative-branch selection are controllable, whereas all the other transitions are uncontrollable under the controller-synthesis mechanisms provided by the Romeo tool.

Strategy for alternative nodes. All alternative transitions have interval $[0, 0]$, so σ (III) decides which branch to authorize when the alternative becomes enabled, rather than when to fire it. Once selected, the other branches are permanently disabled. Thus, restricted to A, a winning strategy selects exactly one outgoing branch per alternative node, directly defining the concrete task $\bar{\tau}$ from τ (Section II-B2).

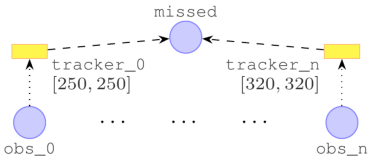


Fig. 3. Deadline observer for a task set τ.

D. Scheduling and Deadline Observation

The scheduler is encoded by functions attached to scheduler transitions. In the considered setting, these functions implement global fixed-priority selection among ready nodes that can execute on engines with the required tag.

This design keeps the scheduling policy isolated from the rest of the TPN structure: supporting a different policy only requires changing the guard and update functions attached to scheduler transitions, not the net structure itself.

To verify schedulability, we introduce an auxiliary TPN component called an observer. Figure 3 illustrates the observer model for a task set composed of n tasks. For each task τ_i, the observer contains a timed transition `tracker_i`, whose firing time corresponds to the relative deadline D_i and its clock starts when a new job of τ_i is released. If the deadline transition becomes fireable before completion of the corresponding job, a place named `missed` is marked. The presence of a token in this place indicates a deadline miss. Schedulability analysis is therefore reduced to a reachability problem in the TPN.

In classical verification, the objective is to determine whether a state satisfying `missed > 0` is reachable. In our case, since alternative selections are modeled using controllable transitions in Romeo, the analysis instead checks whether there exists a controller strategy such that all executions avoid states satisfying `missed > 0`.

V. EXPERIMENTS

This section evaluates the proposed approach. The evaluation is organized as follows. We first describe the synthetic task-set generation protocol and the experimental platform. We then

study the impact of task-set size on the scalability of the proposed approach, with a particular focus on verification time and the behavior of the state-space exploration.

A. Synthetic Experiments and Task-Set Generation

The experiments are conducted on a server equipped with two Intel Xeon E5-2620 processors, each with 6 physical cores and Hyper-Threading enabled, for a total of 24 logical processors, and 128 GB of RAM. Since each Romeo verification run is executed sequentially, the reported execution times mainly reflect the cost of state-space exploration rather than parallel execution effects. For each verification run, we record the schedulability verdict and the time required to obtain it.

To evaluate the impact of task-set size, we generate synthetic HPC-DAG task sets with increasing numbers of computation nodes. We consider five task-set size levels, namely 16, 32, 40, 60, and 80 computation nodes, with 50 randomly generated task sets for each size level. The task periods, execution times, relative deadlines, and workload distributions among tasks and computation nodes are randomly generated for each task set.

The generated task sets are evaluated on a heterogeneous execution platform composed of 6 CPU cores, 2 discrete GPUs (dGPUs), and 2 Deep Learning Accelerators (DLAs). Computation nodes are assigned to compatible execution-engine types according to the heterogeneous execution model described in (section II). Each generated task set is then translated into the corresponding Time Petri Net model and verified using the Romeo model checker.

Each verification run is limited to 600 s. Runs without a verdict within this limit are classified as inconclusive and excluded from the average verification time, but retained in the scalability analysis.

B. Impact of Task-Set Size

Figure 4 shows the average verification time as a function of task-set size. Verification is almost instantaneous for task sets containing 16 subtasks, but increases to approximately 165 s for 32 subtasks and about 450 s for 40 subtasks. For larger task sets containing 60 and 80 subtasks, most runs reach the 600 s timeout.

This degradation results from the growing state space induced by concurrency, timing constraints, heterogeneous resources, and alternative execution paths, making exhaustive controller-synthesis verification the main scalability bottleneck.

Compared with existing HPC-DAG schedulability analyses [6], [7], which provide scalable sufficient tests but may return pessimistic negative verdicts, our approach provides exact verification for the modeled TPN. To the best of our knowledge, this is the first formal-methods-based approach to provide exact schedulability verification for HPC-DAG task sets. A positive result therefore establishes the existence of a valid alternative selection, while the reported verification time reflects the cost of exhaustive state-space exploration.

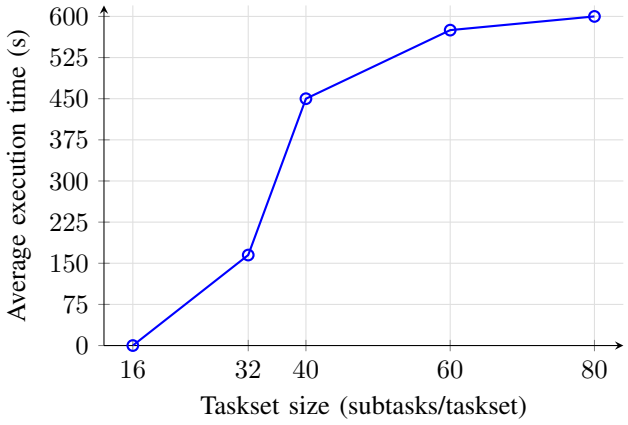


Fig. 4. Verification time on task sets with increasing numbers of subtasks.

VI. Conclusion and Future Work

This work presented a TPN model for the schedulability analysis of heterogeneous HPC-DAG task sets. The model is well suited to the HPC-DAG semantics because it separates controllable design-time alternatives from uncontrollable runtime behavior, while preserving timing constraints and scheduler decisions. The approach was implemented and tested with the Romeo tool on synthetic task sets. These experiments confirmed the relevance of the modeling approach, but also showed that direct state-space exploration becomes limited for large task sets with many alternatives.

Future work will therefore focus on integrating schedulability tests inside the TPN-based checking process. Utilization constraints, critical-path conditions, and response-time based tests will be used to detect infeasible configurations earlier and accelerate verification on larger heterogeneous workloads.

References

[1] A. Saifullah, D. Ferry, J. Li, K. Agrawal, C. Lu, and C. D. Gill, "Parallel real-time scheduling of DAGs," *IEEE Transactions on Parallel and Distributed Systems*, vol. 25, no. 12, pp. 3240–3252, 2014.

[2] J. Li, J.-J. Chen, K. Agrawal, C. Lu, C. D. Gill, and A. Saifullah, "Analysis of federated and global scheduling for parallel real-time tasks," in *Proceedings of the 26th Euromicro Conference on Real-Time Systems*, 2014, pp. 85–96.

[3] A. Melani, M. Bertogna, V. Bonifaci, A. Marchetti-Spaccamela, and G. C. Buttazzo, "Schedulability analysis of conditional parallel task graphs in multicore systems," *IEEE Transactions on Computers*, vol. 66, no. 2, pp. 339–353, 2017.

[4] S. Baruah, "The federated scheduling of systems of conditional sporadic DAG tasks," in *Proceedings of the International Conference on Embedded Software*, 2015, pp. 1–10.

[5] M. Nasri and B. B. Brandenburg, "An exact and sustainable analysis of non-preemptive scheduling," in *Proceedings of the IEEE Real-Time Systems Symposium*, 2017, pp. 12–23.

[6] H.-E. Zahaf, N. Capodieci, R. Cavicchioli, G. Lipari, and M. Bertogna, "The HPC-DAG task model for heterogeneous real-time systems," *IEEE Transactions on Computers*, vol. 70, no. 10, pp. 1747–1761, 2021.

[7] S. Baruah, "Feasibility analysis for HPC-DAG tasks," *Real-Time Systems*, vol. 58, no. 2, pp. 134–152, 2022.

[8] P. M. Merlin, "A study of the recoverability of computing systems," Ph.D. dissertation, University of California, Irvine, 1974.

[9] D. Lime and O. H. Roux, "Expressiveness and analysis of scheduling extended time petri nets," in *IFAC Proceedings Volumes*, vol. 36, no. 13, 2003, pp. 189–197.

[10] D. Lime, O. H. Roux, C. Seidner, and L.-M. Traonouez, "Romeo: A parametric model-checker for petri nets with stopwatches," in *Tools and Algorithms for the Construction and Analysis of Systems*, ser. Lecture Notes in Computer Science, vol. 5505. Springer, 2009, pp. 54–57.

Learning-Assisted Configuration for Deterministic Communications in Time-Sensitive Networks

Mohamed El Amine Chabane*†, Siwar Ben Hadj Said*, Mireille Sarkiss†
*Université Paris-Saclay, CEA List, France
{mohamed-elamine.chabane, siwar.benhadjsaid}@cea.fr
†SAMOVAR, Télécom SudParis, Institut Polytechnique de Paris, France
mireille.sarkiss@telecom-sudparis.eu

***Abstract*—Future cyber-physical applications increasingly require deterministic communication services with bounded latency and predictable behavior. Time-Sensitive Networking (TSN), and IEEE 802.1Qbv, provides mechanisms to enforce deterministic transmissions through cyclic Gate Control Lists (GCLs). However, synthesizing GCLs is a combinatorial scheduling problem whose computational cost increases rapidly with the number of flows and switches, limiting the responsiveness of exact optimization approaches when network configurations must be updated dynamically. This paper investigates a learning-assisted approach for scalable IEEE 802.1Qbv scheduling. We formulate the scheduling problem as a deep reinforcement learning (DRL) task and introduce constraint-aware action masking to prevent infeasible gate decisions. For multi-switch networks, a Multi-Head Deep Q-Network (MH-DQN) factorizes the joint decision space into one scheduling head per switch, reducing the output space from 9^K joint actions to $9K$ outputs for K switches. The inferred slot-level decisions are converted into standard-compliant GCLs and independently evaluated using OMNeT++/INET packet-level simulation. Multi-switch experiments show that the learned scheduler preserves higher TT schedulability than EDF and no-GCL baselines as traffic load increases.**

***Index Terms*—Time-Sensitive Networking, IEEE 802.1Qbv, deterministic communications, Gate Control Lists, reinforcement learning, real-time networks, network configuration.**

I. Introduction

Future cyber-physical systems increasingly rely on communication networks in which timing correctness is as important as successful packet delivery. Industrial control loops, collaborative robots, autonomous machines, and digital twins require bounded end-to-end latency, low jitter, and predictable communication behavior. In such systems, a packet delivered after its deadline may be unusable even if it is eventually received.

Time-Sensitive Networking (TSN) extends standard Ethernet with mechanisms for deterministic communication. In particular, IEEE 802.1Qbv [1] defines the Time-Aware Shaper (TAS), which controls the transmission gates associated with switch egress queues according to a cyclic Gate Control List (GCL). A properly configured GCL creates deterministic transmission windows for critical Time-Triggered (TT) traffic while controlling interference from Best-Effort (BE) traffic.

GCL synthesis is a combinatorial scheduling problem involving flow periods, deadlines, packet sizes, routes, queue assignments, guard bands, and precedence constraints across multiple switches. Satisfiability Modulo Theories (SMT), Integer Linear Programming (ILP), and Constraint Programming (CP) approaches can provide strong feasibility guarantees, but their computation time may increase rapidly with the number of flows, switches, and scheduling variables. This becomes problematic when a configuration must be recomputed after flow admission, topology modification, or industrial reconfiguration.

This need motivates the use of learning-assisted scheduling. Once trained offline, a reinforcement learning (RL) policy can generate scheduling decisions through neural network inference without solving a complete optimization problem from scratch at each reconfiguration. However, applying RL to TSN remains challenging: the scheduling actions must satisfy strict transmission constraints, and a direct representation of all simultaneous multi-switch decisions leads to an exponentially growing action space.

The contributions of this paper are:

- a constraint-aware RL formulation for IEEE 802.1Qbv scheduling in which action masks prevent infeasible scheduling decisions,
- a centralized Multi-Head Deep Q-Network (MH-DQN) architecture that factorizes a 9^K joint action space into $9K$ outputs for a network of K controlled switches
- an end-to-end evaluation pipeline converting learned slot-level decisions into deployable GCLs and validating them independently using OMNeT++/INET packet-level simulation.

The remainder of this paper is organized as follows. Section II reviews the TSN configuration context and related work. Section III presents the scheduling methodology and the proposed MH-DQN architecture. Section IV describes the experimental methodology and evaluates the schedulability of the generated configurations. Section V concludes the paper and discusses future research directions.

II. Background and Related Work

A. TSN and Centralized Configuration

TSN provides a family of IEEE mechanisms for deterministic Ethernet. Time synchronization is provided by standards such as IEEE 802.1AS [2], and traffic shaping and scheduling are handled by mechanisms such as IEEE 802.1Qbv. In

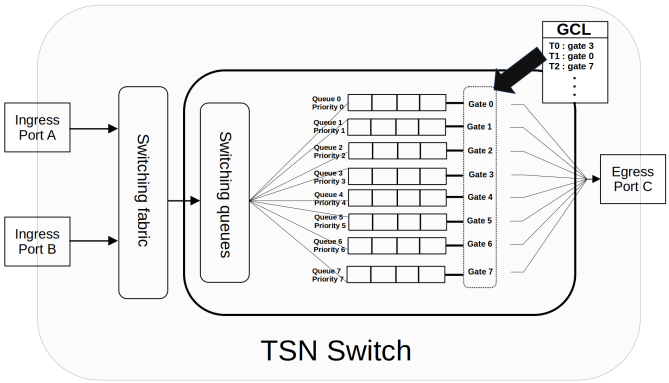


Figure 1. TAS mechanism specified by IEEE 802.1Qbv.

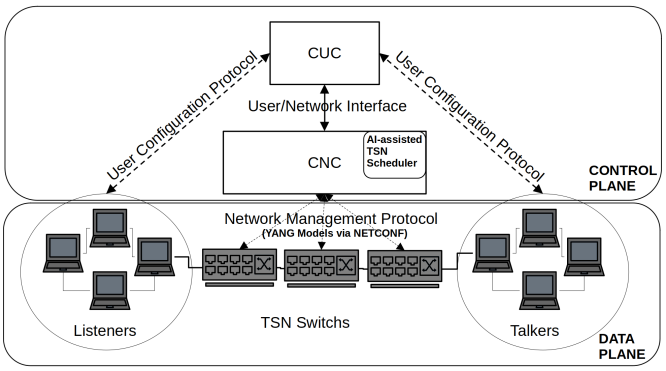


Figure 2. AI-assisted TSN configuration architecture based on the IEEE 802.1Qcc framework.

IEEE 802.1Qbv, each switch output port contains several priority queues, each controlled by a transmission gate (Figure 1). The TAS opens and closes these gates according to a cyclic GCL, creating reserved windows for critical traffic.

The configuration process can be integrated into the centralized model defined by IEEE 802.1Qcc [3]. In this architecture, the Centralized User Configuration (CUC) collects application requirements, and the Centralized Network Configuration (CNC) computes and deploys network configurations. This centralized setting is suitable for learning-assisted scheduling because the CNC can maintain a global view of the topology, traffic requirements, device capabilities, and monitoring information.

Figure 2 illustrates the targeted AI-assisted configuration workflow. The learning-based configuration engine is envisioned as part of the CNC. Given flow requirements and network information, it generates IEEE 802.1Qbv-compatible configurations that can later be translated into management data models and deployed to devices.

B. Existing Scheduling Approaches

The generation of IEEE 802.1Qbv GCLs has been studied using exact, heuristic, and learning-based approaches. Exact methods formulate scheduling as a constraint satisfaction or optimization problem. Craciunas *et al.* [4] proposed a foundational formulation for real-time communication scheduling in TSN, and Steiner *et al.* [5] used array theory encoding for IEEE 802.1Qbv synthesis. Dürr and Nayak [6] studied no-wait packet scheduling to reduce intermediate buffering and jitter. These approaches are valuable because they can provide feasibility guarantees, but they may become expensive on large instances.

To improve scalability, several works have proposed heuristic or hybrid methods for TAS scheduling ([7]–[9]). These methods can generate schedules faster than exact solvers on larger instances. However, their performance is often sensitive to the selected priority rules, the traffic pattern, the topology, and the initial ordering of streams.

Recent learning-based methods, including DeepScheduler [10] or the GCN-TD3 framework [11], have shown the potential of RL for network scheduling, but often do not directly control per-switch GCL decisions under IEEE 802.1Qbv constraints. Our work addresses this gap through centralized multi-head Q-learning with action masking.

III. Learning-Assisted Scheduling Methodology

A. Problem Formulation and Workflow

The input of the scheduling problem consists of the network topology, link capacities, switch capabilities, and a set of critical flows. Each flow is characterized by its period, deadline, packet size, priority, and route. The objective is to generate one IEEE 802.1Qbv-compatible GCL per controlled egress port such that end-to-end timing constraints are satisfied while network resources are used efficiently.

The proposed methodology follows the two-stage workflow illustrated in Fig. 3. During the offline stage, scenarios containing network topologies, traffic flows, and routes are used to train the learning-assisted scheduler in a controllable TSN environment. The scheduler observes the network state, selects gate actions, and receives rewards according to the timing behavior of critical traffic.

During the online stage, the trained policy is executed in deterministic inference mode. The resulting slot-level scheduling decisions are converted into IEEE 802.1Qbv GCL entries and exported to an independent OMNeT++/INET packet-level simulation for validation.

B. Learning-Assisted Scheduling Principle

The scheduling problem is modeled as a finite-horizon Markov Decision Process (MDP). Time is divided into N_{TS} slots of duration Δt over a hyperperiod H (smallest time interval over which the GCL schedule repeats). One hyperperiod of the GCL corresponds to one episode of learning. At each slot or step of learning, the agent observes the network state and selects a gate action. A reward is issued when critical packets complete transmission, according to their delay relative to their deadline. This formulation captures the sequential nature of scheduling decisions.

The state contains, for each switch and priority queue, the queue occupancy and the Head-of-Line packet age. The action is a joint vector specifying, for each switch, whether to remain idle (all gates closed), serve Best-Effort traffic, or open one of the TT priority queues, ie. nine possible actions per switch.

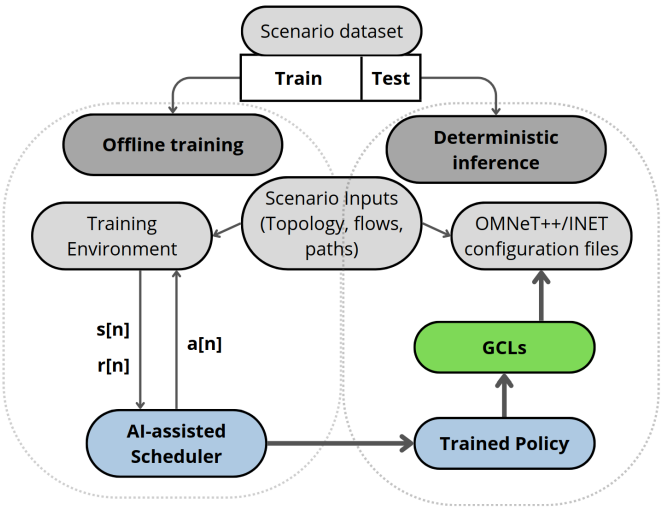


Figure 3. Offline training, deterministic inference, GCL extraction, and independent OMNeT++/INET validation workflow.

The reward is sparse and is generated when TT packets reach their final destinations: each delivered packet receives a value encouraging early end-to-end delivery relative to its deadline and penalizing late transmissions.

C. Constraint Integration

In many learning problems, invalid actions are penalized after they occur. This is not sufficient for deterministic networking, where invalid gate decisions may correspond to physically impossible or non-deployable configurations. Therefore, action masks are used to remove inadmissible decisions before action selection.

Constraints include selecting only non-empty queues, maintaining gate continuity during non-preemptive frame transmission, preventing multiple starts of the same periodic flow in the same period, and reserving guard-band intervals before critical transmissions. This mechanism reduces useless exploration, improves learning stability, and ensures that the generated slot-level decisions can be converted into valid GCL entries.

D. Multi-Switch Scheduling with MH-DQN

The first milestone of our work considered learning-assisted IEEE 802.1Qbv scheduling on a single switch output port [12], [13] using Q-learning algorithm. This allowed the basic MDP formulation, action masking, reward design, and GCL extraction pipeline to be validated in a controlled setting.

Extending the scheduling problem to multiple switches introduces a major scalability issue. A conventional DQN could encode all simultaneous switch decisions as a single joint action. If each switch has nine admissible gate decisions, a network containing K switches would require 9^K output actions, which rapidly becomes intractable.

To avoid this exponential output representation, we introduce a Multi-Head Deep Q-Network (MH-DQN). The architecture maintains a centralized global state representation and a shared neural backbone, and produce one local Q-value head for each controlled switch. Each head outputs nine Q-values corresponding to the local gate decisions. The resulting network therefore requires $9K$ outputs instead of 9^K joint outputs.

The selected local decisions are applied jointly to the network environment. Although the action representation is factorized, all heads are conditioned on the shared global state and optimized using a common end-to-end reward, thereby preserving centralized coordination between switches.

Fig. 4 illustrates this principle through a multi-head learning loop.

The agent observes the global network state, processes it through a shared neural backbone, and produces one scheduling decision head per switch. The environment then applies the selected joint action, updates the observable state, and returns a reward.

IV. Experimental Evaluation

A. Experimental Setup

We evaluate the multi-switch scheduler using OMNeT++/INET on a topology with $K = 4$ switches. The number of TT flows varies from 10 to 100. Flow periods are randomly selected from $\{125\,\mu s, 250\,\mu s, 500\,\mu s, 1\,ms, 3\,ms\}$, with deadlines equal to periods, and packet sizes are selected from $\{64, 128, 256, 512, 1024\}$ bytes. These values provide heterogeneous sub-millisecond and millisecond communication cycles. Background Best-Effort (BE) traffic follows a Poisson process with $\lambda_0 = 3000$ packets/s.

After training, MH-DQN is executed in deterministic inference mode. The generated slot-level decisions are converted into IEEE 802.1Qbv GCLs and independently validated in OMNeT++/INET.

B. Baselines and Metrics

MH-DQN-generated GCLs are compared with GCLs synthesized using a lightweight Earliest Deadline First (EDF) slot-level heuristic and with a no-GCL configuration. EDF provides a computationally inexpensive deadline-aware reference for evaluating the scheduling quality of the learned policy.

The main metric is TT schedulability, defined as the percentage of TT packets received before their end-to-end deadline.

C. Schedulability Results

Figure 5 shows TT schedulability as the number of flows increases. MH-DQN achieves the highest schedulability across the evaluated loads. Its schedulability decreases from 100% at 10 flows to 97% at 100 flows, compared with 100% to 50% for EDF and 100% to 22% without GCLs. The increasing gap under high load indicates that coordinated multi-switch gate decisions become particularly beneficial when contention increases.

V. Conclusion

This paper presented a learning-assisted approach for IEEE 802.1Qbv GCL synthesis based on constraint-aware MH-DQN. By factorizing the joint action space into per-switch

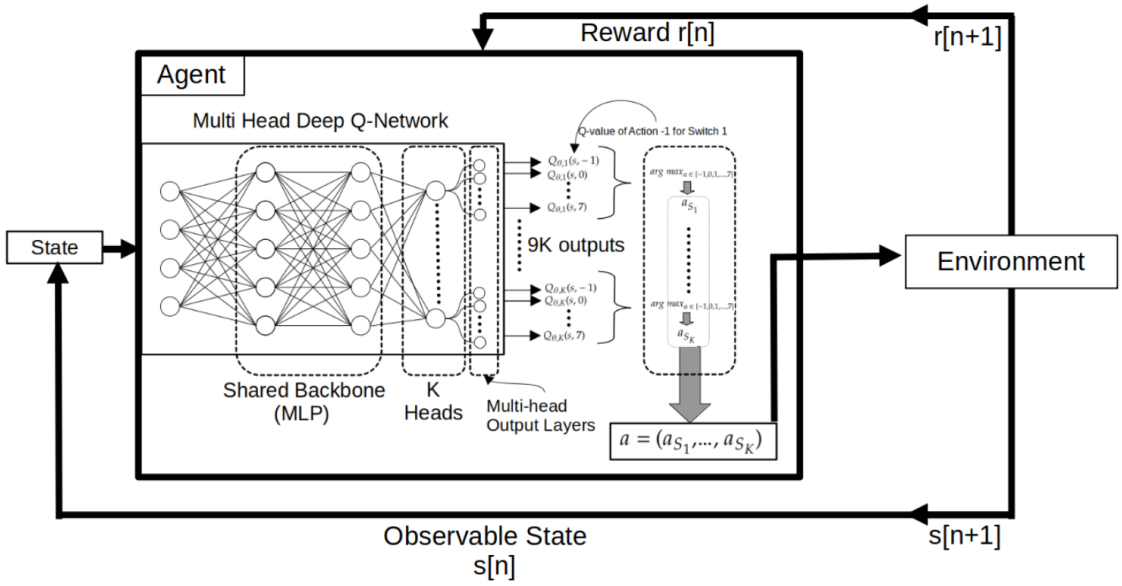


Figure 4. Overview of the proposed multi-head reinforcement learning loop for multi-switch TSN scheduling.

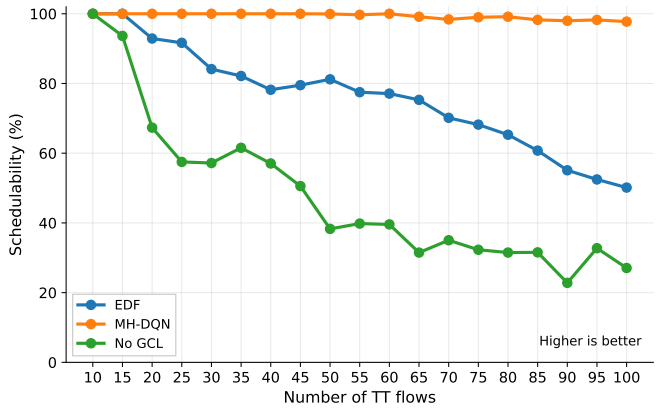


Figure 5. TT schedulability versus the number of TT flows for MH-DQN, EDF, and the no-GCL baseline.

decision heads, the proposed method supports scalable multi-switch scheduling while preserving centralized coordination. The generated schedules are converted into deployable GCLs and validated in OMNeT++/INET.

Results show that MH-DQN maintains high TT schedulability under increasing traffic load, reaching 97% at 100 flows compared with 50% for EDF and 22% without GCLs. Future work will consider larger topologies and a broader benchmark including heuristic, optimization-based, and learning-based schedulers, with joint evaluation of scheduling quality and schedule-generation time.

Acknowledgments

This work was supported by the French National Research Agency (ANR) as part of the France 2030 investment plan under grants ANR-22-PEFT-0007 (NF-FITNESS project) and ANR-24-PEFT-0001 (NF-DONUTS project).

References

[1] IEEE Std 802.1Qbv-2015, *IEEE Standard for Local and Metropolitan Area Networks–Bridges and Bridged Networks–Amendment: Enhancements for Scheduled Traffic*, IEEE, 2015.

[2] IEEE Std 802.1AS-2020, *IEEE Standard for Local and Metropolitan Area Networks–Timing and Synchronization for Time-Sensitive Applications*, IEEE, 2020.

[3] IEEE Std 802.1Qcc-2018, *Stream Reservation Protocol Enhancements and Performance Improvements*, IEEE, 2018.

[4] S. S. Craciunas, R. S. Oliver, M. Chmelik, and W. Steiner, "Scheduling real-time communication in IEEE 802.1Qbv time sensitive networks," in *Proc. RTNS*, 2016, pp. 183–192.

[5] W. Steiner, S. S. Craciunas, and R. S. Oliver, "IEEE 802.1Qbv gate control list synthesis using array theory encoding," in *Proc. RTAS*, 2018, pp. 45–56.

[6] F. Dürr and N. G. Nayak, "No-wait packet scheduling for IEEE time-sensitive networks," in *Proc. RTNS*, 2016, pp. 203–212.

[7] M. Pahlevan, N. Tabassam, and R. Obermaisser, "Heuristic list scheduler for time triggered traffic in time sensitive networks," *ACM SIGBED Review*, vol. 16, no. 1, pp. 15–20, 2019.

[8] M. Vlk, K. Brejchová, Z. Hanzálek, and S. Tang, "Large-scale periodic scheduling in time-sensitive networks," *Computers & Operations Research*, vol. 137, p. 105512, 2022.

[9] D. Bujosa, M. Ashjaei, A. V. Papadopoulos, T. Nolte, and J. Proenza, "HERMES: Heuristic multi-queue scheduler for TSN time-triggered traffic with zero reception jitter capabilities," in *Proc. International Conference on Real-Time Networks and Systems (RTNS)*, 2022, pp. 70–80.

[10] X. He, X. Zhuge, F. Dang, X. Wang, and Z. Yang, "DeepScheduler: Enabling flow-aware scheduling in time-sensitive networking," in *Proc. IEEE INFOCOM*, 2023, pp. 1–10.

[11] S. T. Islam and A. B. Muslim, "AI-based dynamic schedule calculation in Time Sensitive Networks using GCN-TD3," in *Proc. IFIP Networking Conference*, 2024.

[12] M. E. A. Chabane, S. Ben Hadj Said, and M. Sarkiss, "Q-Learning Approach for Gate Control List Scheduling in Time Sensitive Networks," in *Proc. IEEE/IFIP Network Operations and Management Symposium (NOMS)*, 2026, accepted and presented.

[13] M. E. A. Chabane, S. Ben Hadj Said, and M. Sarkiss, "Optimisation de l'ordonnancement IEEE 802.1Qbv dans les réseaux TSN par Q-learning," in *Proc. 28es Rencontres Francophones sur les Aspects Algorithmiques des Télécommunications (ALGOTEL)*.

Supporting Heterogeneous Space Applications on Linux through Container-Level Temporal Isolation

Merlin Kooshmanian
and Frédéric Boniol
ONERA, Toulouse, France
{firstname}.{lastname}@onera.fr

Jérôme Ermont
IRIT, Toulouse, France
{firstname}.{lastname}@irit.fr

Simon Corbin
and Lucas Miné
CNES, Toulouse, France
{firstname}.{lastname}@cnes.fr

***Abstract*—Linux-based platforms are increasingly considered for spacecraft onboard software because they enable the deployment of software capabilities that were traditionally confined to the ground segment, supporting increasingly complex and autonomous missions. However, supporting heterogeneous space applications on Linux requires temporal isolation between containers while preserving different scheduling semantics inside each container. This paper presents an approach combining a bi-class application model, a Linux kernel patch named Task Group Bandwidth Server (TGBS), and a reservation-sizing method. The model distinguishes deadline-constrained Real-Time (RT) tasks from Quality-of-Service (QoS) tasks constrained by minimum execution progress. TGBS enforces CPU reservations at container level while preserving native Linux scheduling policies, and the sizing method provisions bandwidth for both RT guarantees and QoS progress. This framework provides a basis for predictable execution of heterogeneous containerized workloads in Linux-based space architectures.**

I. Introduction

Modern spacecraft flight software is becoming increasingly complex and integrates a growing diversity of onboard functions, including platform control, supervision services, payload management, and payload data processing [1]. To support this evolution while maintaining high levels of safety and dependability, spacecraft architectures increasingly rely on partitioned architectures that isolate software subsystems while allowing them to share common computing resources [2], [3], [4]. Temporal isolation is a key requirement of such architectures, ensuring that the timing behavior of one subsystem remains independent of the execution of the others.

Linux is attracting growing interest for onboard software because of its hardware support, rich software ecosystem, and extensive development tools [5], [6]. Combined with containerization technologies and recent hierarchical CPU reservation mechanisms [7], [8], it enables modular deployment while providing temporal isolation at the container level.

Rather than replacing traditional partitioned architectures, Linux-based platforms should be regarded as a complementary solution. While partitioned RTOS and hypervisors remain well suited for critical onboard functions, Linux enables the deployment of software traditionally confined to the ground segment because of the limited computing resources and software environments historically available onboard, such as autonomous orbit control algorithms or scientific data-processing pipelines. Bringing such applications onboard has the potential to increase spacecraft autonomy and computational capabilities. The challenge is therefore to strengthen its temporal predictability and isolation guarantees while preserving the flexibility and software richness that motivate its adoption.

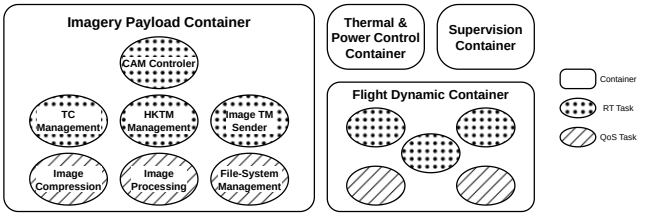


Fig. 1. Partial containerized flight software architecture with a focus on an imagery payload container mixing RT and QoS tasks.

In a Linux-based partitioned architecture, each major spacecraft subsystem can be encapsulated within an independent container. Figure 1 illustrates an imagery payload container combining heterogeneous activities. It hosts control functions, telemetry management, and image transmission together with background services such as image processing, compression, and storage. While the former require bounded response times to satisfy activation deadlines, the latter require sustained access to processing resources to guarantee continuous progress. Providing temporal isolation between containers is a prerequisite for such architectures, but it does not address the heterogeneous execution requirements within each isolated subsystem. Supporting both deadline-constrained and background activities within the same container while preserving predictable execution therefore remains a key challenge.

This paper presents an approach for supporting heterogeneous containerized space applications through container-level temporal isolation. The proposed approach combines hierarchical CPU reservations, an application model distinguishing deadline-constrained real-time activities from progress-oriented background activities, and a reservation-sizing method that allocates sufficient processing capacity while preserving predictable execution. The remainder of the paper first presents the addressed problem, then introduces the proposed application model and solution, before concluding with future research directions.

II. Problem Statement

Providing predictable execution for containerized space applications requires addressing three complementary challenges: enforcing temporal isolation between containers, supporting

heterogeneous execution requirements within each container, and provisioning sufficient computing resources to satisfy these requirements.

Linux already provides many of the required building blocks. It supports multiple scheduling classes, allowing deadline-driven, and fair-share scheduling policies to coexist within the same operating system. However, providing temporal isolation at container level remains challenging. The *Hierarchical Constant Bandwidth Server* (HCBS) Linux kernel patch provides group-level CPU reservations, but its implementation is restricted to real-time tasks [7], [8]. It therefore does not directly support containers combining multiple scheduling semantics. To address this limitation, we previously introduced the *Task Group Bandwidth Server* (TGBS), a Linux kernel patch that enables container-level reservations while allowing tasks inside the container to remain scheduled by their native Linux policies [9], [10].

The literature also proposes several models for workloads that cannot be described using strict hard real-time assumptions. Soft real-time and mixed-criticality models relax deadline requirements through bounded deadline misses, degraded execution modes, or adaptive resource allocation [11], [12]. Workload-oriented programming further characterizes applications according to the amount of work completed over time rather than individual job deadlines [13]. Although these approaches capture different execution semantics, they are not integrated into a unified model suitable for heterogeneous Linux containers.

Finally, hierarchical scheduling has been extensively studied through the Compositional Scheduling Framework (CSF) [14], which provides an analytical basis for dimensioning CPU reservations. CSF and its derivatives have successfully been applied to hierarchical real-time systems and reservation-based container frameworks [15], [16]. Nevertheless, existing formulations generally assume that each component hosts a homogeneous workload analyzed under a single scheduling abstraction.

Taken together, these observations show that the required ingredients already exist independently. Linux provides hierarchical CPU reservations and multiple scheduling policies, workload models describe different execution semantics, and CSF provides analytical reservation sizing. However, no existing approach combines these elements into a unified framework capable of supporting heterogeneous containerized space applications. The remainder of this paper addresses this challenge by introducing an application model together with a reservation mechanism and sizing method for predictable execution.

III. Application Model

Figure 2 summarizes the application model considered in this work. Each spacecraft subsystem is encapsulated within a Linux container associated with a periodic CPU reservation. The reservation allocates a predictable share of processor time to the container, while tasks inside the container execute according to their native Linux scheduling policies. The proposed model distinguishes two complementary task classes: *Real-Time (RT)* tasks, constrained by deadlines, and *Quality-of-Service (QoS)* tasks, constrained by minimum execution progress.

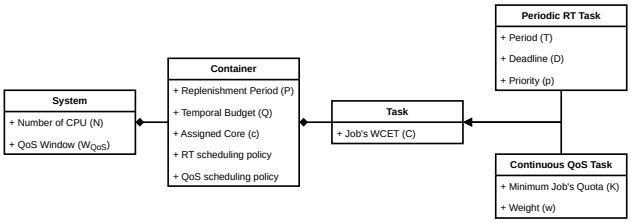


Fig. 2. Overview of the proposed application model.

A. Container Model

Each container is modeled as a periodic CPU reservation following the Constant Bandwidth Server (CBS) abstraction. It is characterized by a temporal budget Q_Π and a replenishment period P_Π, defining a reserved processor bandwidth

$$U_\Pi = \frac{Q_\Pi}{P_\Pi}.$$

A container Π is therefore represented by

$$\Pi = \left(Q_\Pi, P_\Pi, \mathcal{T}_\Pi^{RT}, \mathcal{T}_\Pi^{QoS}\right),$$

where $\mathcal{T}_\Pi^{RT}$ and $\mathcal{T}_\Pi^{QoS}$ denote the sets of hosted RT and QoS tasks, respectively.

CBS enforces the reservation by allowing the container to execute for at most Q_Π units of processor time every period P_Π. Once the reserved budget is exhausted, the container is suspended until its budget is replenished at the beginning of the next period. As a result, the container receives a bounded share of the processor independently of the execution demand of the other containers.

B. Real-Time Task Model

RT tasks model activities whose execution is driven by periodic or event-triggered activations. They follow the conventional periodic real-time task model

$$\tau_i^{RT} = (C_i, T_i, D_i, p_i),$$

where C_i denotes the worst-case execution time, T_i the period, D_i the relative deadline, and p_i the fixed priority within the RT scheduling class.

The timing constraint of an RT task follows the classical hard real-time semantics: every released job must complete before its deadline. Inside the container, RT tasks are scheduled according to the Linux `SCHED_FIFO` policy.

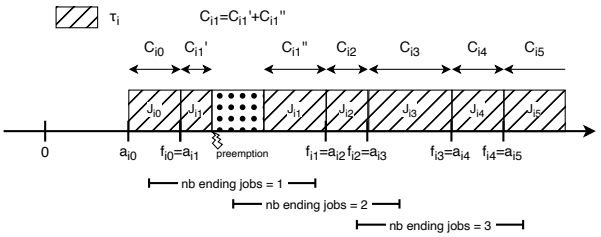


Fig. 3. Continuous QoS task model with immediate job releases and completed-job counting over observation windows. Task τ_i consists of jobs $J_{i,k}$ characterized by their arrival time $a_{i,k}$, finish time $f_{i,k}$, and execution time $C_{i,k}$.

C. Quality of Service Task Model

Unlike RT tasks, some activities are not naturally driven by periodic or event-triggered activations. Instead, they continuously process available data and primarily require sustained execution progress rather than bounded response times.

To model such activities, this work introduces the notion of *QoS task*. A QoS task is modeled as a continuously backlogged task: each new job is released immediately after the completion of the previous one, forming a continuous stream of execution requests, as illustrated in Figure 3.

A QoS task is characterized by

$$\tau_i^{QoS} = (C_i, K_i, w_i),$$

where C_i denotes the execution time of one job, K_i the minimum number of completed jobs required over the QoS observation window W_{QoS}, and w_i the relative weight representing the fraction of the QoS execution capacity allocated to τ_i. Over long intervals, the share of processor time received by τ_i is expected to converge toward w_i.

Since a QoS task is continuously backlogged, its progress requirement can be expressed in terms of processor service. The service required by τ_i^{QoS} to complete its quota over one observation window is

$$S_i = K_i C_i.$$

Unlike RT tasks, whose timing constraint is defined by deadlines, the timing constraint of a QoS task is expressed as a minimum execution progress. Let $N_i(W)$ denote the number of jobs completed over an observation window of duration W. The QoS constraint is therefore defined as

$$N_i(W_{QoS}) \geq K_i.$$

Since QoS tasks remain continuously backlogged, a priority-based scheduler could indefinitely delay low-priority tasks. They are therefore scheduled using the Linux `SCHED_OTHER` scheduling class, which distributes the processor according to configurable weights. The weight w_i plays a role analogous to the priority of RT tasks: while priorities define the execution order of RT activities, weights determine the long-term share of processor time allocated to QoS activities.

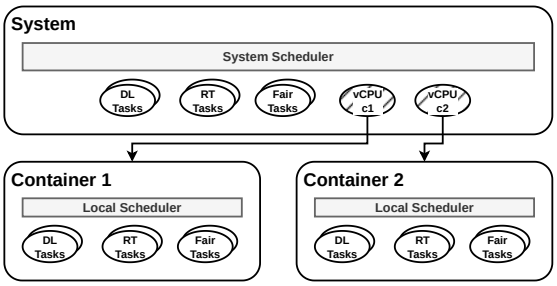


Fig. 4. Principle of TGBS. At the system level, container virtual CPUs compete with native Linux tasks. Each virtual CPU is backed by a CBS deadline server and hosts an independent Linux scheduler supporting the native deadline (DL), real-time (RT), and fair scheduling classes.

IV. Proposed Framework

Building upon the proposed application model, the proposed framework combines two complementary mechanisms. First, a Linux scheduling infrastructure enforces CPU reservations while preserving the native scheduling policies inside each container. Second, a reservation-sizing method determines the amount of processor bandwidth required to satisfy both RT and QoS timing constraints.

A. Linux Reservation Mechanism

Linux provides CPU reservations through its deadline scheduling class, associated with the `SCHED_DEADLINE` policy and based on Constant Bandwidth Server (CBS) semantics. Recent Linux kernels also introduce *deadline servers*, which extend this mechanism by allowing a CBS reservation to execute a group of tasks instead of a single deadline task.

To support heterogeneous containerized applications, this work leverages TGBS [1], that generalizes the HCBS principle by using deadline servers to enforce hierarchical CPU reservations over complete Linux scheduling hierarchies rather than over a single scheduling class. Instead of attaching a deadline server to a single scheduling class, TGBS uses it as a *virtual CPU* representing the whole container at the system level. As illustrated in Figure 4, this virtual CPU hosts an independent Linux scheduling hierarchy, including the deadline, real-time, and fair scheduling classes. Tasks are therefore selected inside the container according to standard Linux semantics, while the associated deadline server enforces the CPU reservation allocated to the subsystem. In other words, TGBS virtualizes the Linux scheduling hierarchy inside each container: from the task perspective, execution follows the same semantics as on the host system, while from the system perspective, the whole container is accounted for and throttled as a single entity.

B. Reservation Sizing

Figure 5 illustrates the intuition behind the sizing method. The container reservation is viewed as a bucket, RT tasks as rigid chunks, and QoS tasks as grains of sand. The bucket is first chosen large enough to contain the RT chunks, which preserves their deadline constraints. However, because these chunks are rigid and the sizing is worst-case based, unused spaces remain inside the bucket. These spaces correspond

[1] Patch releases are archived at https://doi.org/10.5281/zenodo.22108568.

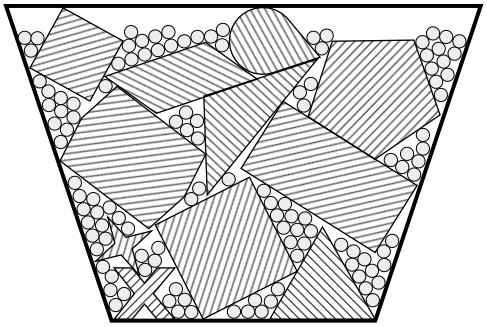


Fig. 5. Bucket analogy for reservation sizing. The bucket represents the container reservation, the large hatched shapes represent RT tasks, and the small grey circles represent QoS demand split into finer-grained execution units. In this example, the reservation selected for the RT workload leaves enough unused capacity to accommodate all QoS demand.

to reservation overhead that can be exploited by QoS tasks. Since QoS demand is less structurally constrained, it can be progressively spread into these gaps. If all QoS grains fit, the reservation is sufficient; otherwise, a larger bucket, i.e., a larger reservation, must be selected.

The sizing method formalizes this intuition by first considering the RT workload alone. Let $(Q_\Pi^{\mathrm{RT}}, P_\Pi^{\mathrm{RT}})$ denote a reservation obtained from a classical CSF-based analysis of $\mathcal{T}_\Pi^{\mathrm{RT}}$, such that all RT tasks meet their deadlines. This reservation defines the initial bucket required by the RT workload.

The processor bandwidth required by the RT and QoS workloads is then estimated as

$$U_\Pi^{\mathrm{RT}} = \sum_{\tau_i \in \mathcal{T}_\Pi^{\mathrm{RT}}} \frac{C_i}{T_i}, \qquad U_\Pi^{\mathrm{QoS}} = \sum_{\tau_i \in \mathcal{T}_\Pi^{\mathrm{QoS}}} \frac{S_i}{W_{\mathrm{QoS}}}. \tag{1}$$

The total bandwidth required by the bi-class workload is therefore

$$U_\Pi = U_\Pi^{\mathrm{RT}} + U_\Pi^{\mathrm{QoS}}. \tag{2}$$

The final reservation keeps the RT replenishment period and increases the temporal budget only if the QoS workload requires additional bandwidth:

$$P_\Pi = P_\Pi^{\mathrm{RT}}, \quad Q_\Pi = \max\left(Q_\Pi^{\mathrm{RT}}, U_\Pi P_\Pi^{\mathrm{RT}}\right), \quad Q_\Pi \le P_\Pi. \tag{3}$$

This rule preserves the RT-only reservation when its unused capacity is sufficient to absorb QoS demand. Otherwise, the container budget is increased until the reservation provides enough total bandwidth for both RT execution and QoS progress. Under ideal weighted sharing between QoS tasks and sufficiently long QoS observation windows, this construction can be shown to preserve RT guarantees while satisfying QoS progress constraints. The corresponding proof is not included here and is left to the full analytical presentation of the method.

V. Conclusion

This paper presented an approach for supporting heterogeneous space applications on Linux through container-level temporal isolation. The proposed model captures two complementary execution requirements within the same container: RT tasks constrained by deadlines and QoS tasks constrained by minimum execution progress. Based on this model, the approach combines TGBS, a Linux kernel patch that virtualizes the Linux scheduling hierarchy inside reserved containers, with a reservation-sizing method that provisions CPU bandwidth for both RT guarantees and QoS progress.

This work provides a first step toward predictable execution of heterogeneous containerized workloads on Linux-based space platforms. Future work will focus on consolidating the analytical guarantees of the sizing method, quantifying the gap between ideal QoS sharing and Linux fair scheduling, and further characterizing the TGBS kernel patch through comparisons with traditional isolation solutions on a representative flight software use case.

References

[1] J. Eickhoff, *Onboard Computers, Onboard Software and Satellite Operations*, 1st ed., ser. Springer Aerospace Technology. Berlin, Heidelberg: Springer Berlin, Heidelberg, 2012.

[2] J. Windsor and K. Hjortnaes, "Time and space partitioning in spacecraft avionics," in *2009 Third IEEE International Conference on Space Mission Challenges for Information Technology*, 2009, pp. 13–20.

[3] J. Windsor, M.-H. Deredempt, and R. De-Ferluc, "Integrated modular avionics for spacecraft — user requirements, architecture and role definition," in *2011 IEEE/AIAA 30th Digital Avionics Systems Conference*, 2011, pp. 8A6–1–8A6–16.

[4] J. Galizzi, J.-J. Metge, P. Arberet, E. Morand, F. Vigeant, A. Crespo, M. Masmano, J. C. Parada, I. Ripoll, V. Brocal *et al.*, "Lvcugen (tsp-based solution) and first porting feedback," in *Embedded Real Time Software and Systems (ERTS2012)*, 2012.

[5] H. Leppinen, "Current use of linux in spacecraft flight software," *IEEE Aerospace and Electronic Systems Magazine*, vol. 32, no. 10, pp. 4–13, 2017.

[6] J. Corbet, "Space-grade linux," https://lwn.net/Articles/1036168/, Feb. 2025, accessed: 2026-05-20.

[7] L. Abeni, A. Balsini, and T. Cucinotta, "Container-based real-time scheduling in the linux kernel," *SIGBED Rev.*, vol. 16, no. 3, p. 33–38, Nov. 2019.

[8] Y. Andriaccio, L. Abeni, and M. Torquati, "Scheduling iot applications in real-time control groups," in *2025 21st International Conference on Distributed Computing in Smart Systems and the Internet of Things (DCOSS-IoT)*, 2025, pp. 01–08.

[9] M. Kooshmanian, F. Boniol, J. Ermont, S. Corbin, and L. Miné, "Towards multi-policy hierarchical scheduling in linux for containerized space applications," in *ERTS 2026 - Embedded Real Time Systems*, ser. ERTS 2026 - Embedded Real Time Systems, Toulouse, France, Feb. 2026.

[10] M. Kooshmanian, "Task group bandwidth server (tgbs)," Software. [Online]. Available: https://github.com/mkooshmanian/TGBS

[11] J. P. Erickson and J. H. Anderson, *Soft Real-Time Scheduling*. Singapore: Springer Nature Singapore, 2022, pp. 233–267.

[12] A. Burns and R. I. Davis, "A survey of research into mixed criticality systems," *ACM Comput. Surv.*, vol. 50, no. 6, Nov. 2017.

[13] S. S. Craciunas, C. M. Kirsch, H. Röck, and A. Sokolova, "Real-time scheduling for workload-oriented programming," Department of Computer Sciences, University of Salzburg, Salzburg, Austria, Tech. Rep. Technical Report 2008-02, Sep. 2008.

[14] I. Shin and I. Lee, "Compositional real-time scheduling framework," in *25th IEEE International Real-Time Systems Symposium*, 2004, pp. 57–67.

[15] V. Struhár, S. S. Craciunas, M. Ashjaei, M. Behnam, and A. V. Papadopoulos, "React: Enabling real-time container orchestration," in *2021 26th IEEE International Conference on Emerging Technologies and Factory Automation (ETFA)*, 2021, pp. 1–8.

[16] V. Struhár, S. S. Craciunas, M. Ashjaei, M. Behnam, and A. V. Papadopoulos, "Hierarchical resource orchestration framework for real-time containers," *ACM Trans. Embed. Comput. Syst.*, vol. 23, no. 1, Jan. 2024.

Controlling Data Flows in Distributed Synchronous Logical Execution Time: the Control Depth Reading Approach

Sid Ahmed Issolah*†, Marc Boyer†, Damien Chabrol*, Guillaume Phavorin*
*Asterios Technologies, Massy, France
{sid-ahmed.issolah, damien.chabrol, guillaume.phavorin}@asterios-technologies.com
†ONERA/DTIS, Université de Toulouse, Toulouse, France
{sid_ahmed.issolah, marc.boyer}@onera.fr

***Abstract*—The synchronous Logical Execution Time (sLET) paradigm makes the behavior of a real-time application independent of the platform it runs on — a property we call transparency, which greatly simplifies its development, deployment, and maintenance. When the application is distributed over several computers, however, communication crosses a network and takes a non-negligible time, which challenges the property. To preserve transparency in this distributed setting, we introduce the Logical Transmission Time Window (LTTW), a logical-time abstraction of the network transmission, and the Control Depth Reading (CDR) approach, which lets each consumer read a slightly older value of a data flow — old enough to give the transmission time to complete — thereby tolerating the network delay while controlling the data age and leaving the logical design unchanged.**

I. Introduction

Safety-critical real-time systems, such as those found in avionics, space, and automotive domains, have become increasingly complex over the last decades [1]. They now integrate a large number of functions, from control and navigation to monitoring, fault management, and advanced assistance features. This evolution has naturally increased the amount of computation these systems must perform. To meet this growing need, critical embedded architectures have historically moved from centralized implementations toward distributed ones, where several computing nodes cooperate through a communication network. However, distribution comes at a deployment cost: designers must take into account the location of execution. An expected property in this context is distribution *transparency*. A system is transparent when the observable behavior of an application is independent of the platform on which it is executed — the same logical application produces the same behavior whether it runs on a centralized platform or is distributed over several nodes. This property is particularly valuable in critical systems: it lets engineers reason about the logical behavior of the application independently of the target architecture, and it simplifies design, deployment, and maintenance.

At the same time, safety-critical real-time systems must satisfy strict timing constraints. In particular, data exchanged between functions must respect end-to-end latency constraints: a value produced by one function must reach its consumers within a bounded delay [2]. Ensuring these timing guarantees is already a central issue in distributed real-time systems; preserving transparency while ensuring them is even more difficult, because network communication introduces a non-negligible delay between producers and consumers.

Contribution. This paper addresses the distribution of synchronous Logical Execution Time (sLET)-based applications while preserving transparency. We work within the sLET paradigm, which expresses LET on logical clocks and is transparent, and we propose the *Control Depth Reading* (CDR) approach. Building on the *Logical Transmission Time Window* (LTTW), a logical-time abstraction of the network transmission, CDR lets a consumer read a slightly older value of a data flow — one that has had time to cross the network — so that the network delay is tolerated while controlling the data age, and without modifying the logical design of the application.

Outline. This paper is structured as follows. Section II presents the synchronous Logical Execution Time (sLET) paradigm. Section III identifies the specific challenges of applying sLET within distributed systems, motivating the definition of the LTTW in Section IV. Section V then presents the CDR approach and how it is applied. Section VI discusses related work, and Section VII concludes.

II. The synchronous Logical Execution Time (sLET) paradigm

sLET paradigm is a real-time abstraction model whose formal foundation, based on synchronous clocks, has been defined in [3]. The sLET paradigm builds on the Logical Execution Time (LET) model. First introduced by Giotto [4], it was later formalized as LET [5]. LET is a real-time abstraction model developed for real-time control applications. It defines a LET window which fixes, for each computation, the logical instants at which it reads its inputs and makes its outputs visible, independently of the actual execution — a principle that sLET shares and extends with logical clocks. Several languages implement the LET and sLET paradigms, and a comparison of these languages has been carried out in [6].

In sLET, the unit of execution is the agent, which corresponds to what the scheduling community calls a task. An agent is a sequence of elementary actions (EAs), the equivalent of jobs, whose activations are triggered by ticks of a logical clock. Each EA executes within a sLET window bounded by two such ticks, an earliest start and a latest finish date. The sLET window abstracts the real execution time of the

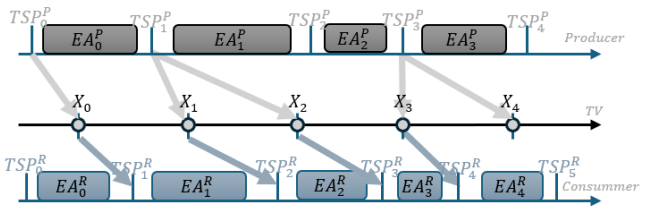


Fig. 1. Communication in sLET through a temporal variable. A produced value updates the sample that comes just after the end of the producer's window, and a consumer reads the sample available just before the start of its own window; the data flow thus depends only on the logical timing.

computation, which must fit within the window [5]. This decouples the logical timing of a computation from its physical execution: an EA reads its inputs from, and makes its outputs visible to, other EAs at its window boundaries — that is, on well-defined logical-clock ticks — independently of when the computation actually runs.

A. Communication in sLET

Communication between agents is mediated by temporal variables (TVs) [7]: shared variables sampled periodically on logical clocks. Each TV is associated with a logical clock, possibly distinct from those of its producer and its consumers. The way these samples are exchanged is governed by the *Visibility Principle* [8], illustrated in Figure 1:

- a value produced by an agent becomes available only from the end of its sLET window, and thus will be sampled on the next tick of the TV's logical clock (greater or equal to the producer's EA deadline);
- a consuming agent reads the sample available at the start of its own sLET window.

This is what makes sLET transparent. Because the instants of production and consumption are fixed on logical clocks, the data flow and the timing are both deterministic: each consumer reads a well-identified sample, regardless of when computations actually run. The behavior of the application is thus correct by construction and independent of the physical platform: the logical design is preserved whatever the underlying execution, which is exactly what we call transparency.

III. sLET in a Distributed System

The transparency of sLET rests on one assumption about communication, the *zero-logical-time* assumption [9]: communication between agents takes place in zero logical time — when an agent terminates its window, the value it produces is immediately available to any consumer at the same logical instant. This must be understood through the sampling of temporal variables. As shown in Figure 1, the value produced at the end of the producer's window is not read directly by the consumer: it is first sampled by the TV, on a clock that is not necessarily harmonic with the producer's and the consumer's windows, so the sampling instant does not, in general, coincide with the end of the producer's window or the start of the consumer's. What the assumption states is that the transmission itself adds no logical delay: the value is available for sampling as soon as it is produced, and the consumer reads, at the start of its window, the sample the Visibility Principle prescribes.

A. The problem under distribution

In a centralized setting, this assumption is easy to support: the producer and the consumer reside on the same hardware and exchange data through a shared memory. The situation changes when the application is distributed. The producer and the consumer now reside on distinct computers connected by a network, and communication is no longer a local access but a network transmission, carried by a resource distinct from the computers themselves. Consider, in Figure 1, the communication that feeds EA_4^R: the sample it reads was produced by EA_2^P and made available at the end of the producer's window. On a single node this sample is available immediately; over a network it must travel from one computer to another between the production of the value and its reading, a logical gap that leaves only a limited budget for the transmission. If this budget is too short for the worst-case transmission time, the sample has not yet arrived when EA_4^R starts, and the consumer reads an older value than the Visibility Principle dictates. The logical behavior is no longer preserved, and with it transparency is lost: the outcome now depends on the actual physical transmission time.

The question is therefore how to preserve the logical model, and its zero-logical-time assumption [10], once transmission takes a non-negligible physical time — without giving up transparency.

IV. The Logical Transmission Time Window

To reason about network transmission within the logical model, we introduce the Logical Transmission Time Window (LTTW): just as the sLET window abstracts the physical execution time of a computation, the LTTW abstracts the physical transmission time of a communication into a logical interval. Formally, consider a message m sent by a producer to a consumer. Its transmission is a physical event whose start and completion instants, s_m and f_m, are not known in advance; all that is known is that it lasts no more than the worst-case transmission time $WCTT_m$:

$$f_m - s_m \leq WCTT_m. \tag{1}$$

To abstract this transmission within the logical model, we enclose it in a logical interval, the LTTW:

$$LTTW_m = [a_m, b_m], \tag{2}$$

whose bounds, a_m and b_m, denote the left and right bounds of the interval, respectively, are deduced from the logical design. They constrain the transmission to take place within them:

$$a_m < s_m < f_m < b_m. \tag{3}$$

The transmission is then correctly abstracted — and transparency preserved — as soon as the worst-case transmission time fits within the window:

$$WCTT_m < |LTTW_m| = b_m - a_m. \tag{4}$$

The bound $WCTT_m$ is the result of a network analysis (e.g. Network Calculus [11]), which is outside the scope of this paper. The endpoints a_m and b_m, for their part, are

set according to the distribution strategy chosen to handle the transmission — for instance encapsulating it within the producer window, as in transparent LET distribution [9]. The LTTW is thus a general abstraction whose concrete definition depends on this strategy; for space reasons, we do not detail these strategies here. A similar notion appears, under different names, in SL-LET [10], [12] and in LETT [13]; there, however, the interval is encoded into the logical architecture rather than deduced from it. A defining feature of the LTTW is precisely that it is deduced from the existing sLET specification — in particular from the placement of producer and consumer windows — and not added to it: it introduces no new window and modifies no existing one. This is what makes it possible to reason about distributed communication while preserving the original logical design.

V. The Control Depth Reading Mechanism

A. Reading depth in sLET

Beyond the latest sample, sLET allows a consumer to access a bounded history of past values of a temporal variable. Consider, in Figure 2, the consumer EA_3^R reading the temporal variable X: reading the most recent sample gives it the value produced by EA_2^P, whereas reading one sample earlier gives it the value produced by EA_1^P. When the code of an agent refers to the variable X, it is dynamically linked to the last value of the sample, denoted X_s, e.g., X_2 for EA_3^R in Figure 2. However, the programmer may also want to read previous values, denoted $X_{s-\delta}$. This feature exists in sLET, and δ is called the *reading depth* [7], [14]. To help the compiler manage memory, the programmer must define a maximal depth δ_M. The reading depth thus selects how far back in the history the consumer reads, from the current sample ($\delta = 0$) up to δ_M declared by the consumer. This mechanism already exists in sLET, independently of any distribution concern. This ability to reach back into the history of a variable is the mechanism Control Depth Reading builds upon.

B. Control Depth Reading (CDR) strategy

CDR turns the reading depth into a simple contract between the designer and the network: a consumer commits never to read a value fresher than $X_{s-\delta}$, that is, to read a value at least δ ticks of the TV's clock old. This commitment gives the corresponding sample that much additional time to cross the network. Concretely, increasing δ moves the lower bound a_m of the window back in time while its upper bound b_m is unchanged, so the window $LTTW_m(\delta)$ grows with δ: the larger the depth, the larger $|LTTW_m(\delta)|$ available to absorb the worst-case transmission time. This duality between the reading depth and the size of the LTTW is illustrated in Figure 2. For each communication, CDR determines the *minimum reading depth* δ_m for which the transmission fits within the window:

$$\delta_m = \min\{\,\delta \mid WCTT_m < |LTTW_m(\delta)|\,\}. \qquad (5)$$

Hence, reading at any depth $\delta \geq \delta_m$ tolerates the network delay.

The two bounds on the depth have different natures. The minimum depth δ_m is network-driven — imposed by the worst-case transmission time — but it also has a functional

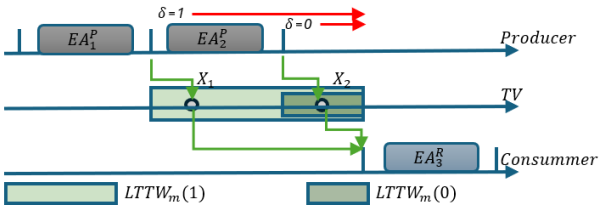


Fig. 2. Duality between reading depth and LTTW. Reading an older sample (depth $\delta = 1$) yields a larger window $LTTW_m(1)$ than reading the latest one (depth $\delta = 0$), leaving more room to absorb the transmission delay.

impact, since the age it introduces enters the response-time analysis of the functional chain. The maximum depth δ_M, in contrast, is a *functional* choice of the application: it reflects how much past history the algorithm needs and how old a value it can still use. For instance, an application computing a moving average of a temperature or a speed deliberately keeps several past samples, regardless of any network consideration. A communication is feasible when the network-driven minimum fits within this functional bound, i.e. when $\delta_m \leq \delta_M$. CDR thus adds a single network-related element to the consumer specification: the minimum depth δ_m required to tolerate the transmission. The rest of the logical model — agents, periods, temporal variables, communication ports — remains unchanged.

C. Age and transparency

The Visibility Principle itself is unchanged: a value is still visible at its production instant. What CDR relaxes is the zero-logical-time assumption — the visible value is no longer required to be *immediately* available, since a reading depth δ specifies an age of consultation that leaves time for the transmission, without affecting functional consistency. Bounding δ within $[\delta_m, \delta_M]$ thus bounds the logical age of the consumed sample. This is a logical age bound, not a physical freshness guarantee: if a producer does not update its variable for several ticks of the TV's clock, the consumer reads the same value again, but its position in the production history remains well-defined. CDR preserves transparency in the following sense: once a consumer commits to a reading depth δ, its observable behavior is identical on every platform whose worst-case transmission time satisfies $WCTT_m < |LTTW_m(\delta)|$. The same application can therefore be deployed on a single node or distributed over several nodes, and it will behave identically as long as the transmission time is respected. The logical architecture — windows, agent periods, temporal variables, communication ports — is never modified; the network delay is absorbed by the controlled reading depth, and the consumer always reads a well-identified sample of bounded and explicitly controlled age.

CDR trades data freshness for communication feasibility. The selected depth must therefore remain compatible with application-level requirements, such as admissible data age and end-to-end latency; assessing its functional impact remains the designer's responsibility. For a fixed δ_M, CDR introduces no additional buffering, since it only selects another sample from the history already maintained by sLET. Finally, CDR relies on a safe upper bound on the $WCTT$: an underestimated bound

may invalidate transparency, whereas a pessimistic one may unnecessarily increase the reading depth.

D. Applicability

CDR can be exploited by a tool in two complementary ways. As a *checker*, the tool is used as a validation aid: the logical design is already given, including the reading depths chosen by the designer, and the tool verifies, for each communication, that the selected depth yields a window large enough to absorb the worst-case transmission time, i.e. that $WCTT_m < |LTTW_m(\delta)|$ holds. As a *solver*, the tool is used as a design aid: the reading depths are not fixed in advance, and the tool derives, from the worst-case transmission times, the minimum depth δ_m that makes the corresponding window large enough. These two aspects combine into a single development process. Starting from a logical design and a network configuration, the checker is first run on the reading depths chosen by the designer. If every communication passes, the distributed design is validated and the process stops. Otherwise, the solver is run on the failing communications, computing the minimum depth δ_m each one requires; the design is updated with these depths, and the checker is run again to confirm that the whole design now holds. In all cases, the final decision rests with the architect, who validates that the resulting reading depths are compatible with the application.

VI. Related Work

The distribution of LET-based applications has, in turn, been studied in several works. A transparent solution was first proposed in TDL (Timing Definition Language) [9]: the network transmission is encapsulated within the existing windows, so that the logical design is preserved. The same encapsulation principle has since been applied to sLET in an industrial project, a Landing Gear System [15]. These approaches preserve transparency but only handle short transmission delays, since the transmission delay must fit within the existing windows.

Other works handle larger delays at the cost of modifying the design. System-Level LET (SL-LET) [10] introduces an explicit logical transmission window; despite the similar terminology, it differs from our LTTW in that it adds a dedicated transmission task to the design. Logical Execution and Transmission Time (LETT) [13] goes further and transforms the LET task itself into a task that embeds the communication. Both modify the logical model and therefore do not preserve transparency.

CDR differs from all of these: it neither restricts itself to short delays nor modifies the design. By exploiting the reading depth already present in sLET to enlarge the logical window, it handles larger delays while keeping the logical design unchanged. Moreover, it can also be combined with the encapsulation approach used in TDL in a hybrid scheme.

VII. Conclusion & perspectives

We addressed the distribution of synchronous Logical Execution Time applications while preserving transparency. sLET guarantees transparency on a single node, but its zero-logical-time assumption is challenged once communication crosses a network. We introduced the Logical Transmission Time Window, a logical-time abstraction of the transmission deduced from the existing design rather than encoded into it, and the Control Depth Reading approach, which turns the reading depth into a contract between the designer and the network: by committing to read a value of bounded age, a consumer gives the transmission enough time to complete. As a result, the same application behaves identically across platforms as long as the worst-case transmission time fits within the corresponding window, and its logical design is never modified. Future work will implement the network checker and the associated solver, characterize the worst-case transmission time through network analysis, and validate the approach on representative case studies.

References

[1] F. Boniol, "New challenges for future avionic architectures," in *Modeling Approaches and Algorithms for Advanced Computer Applications*, ser. Studies in Computational Intelligence. Cham: Springer International Publishing, 2013, vol. 488, pp. 1–1.

[2] M. Dürr, G. von der Brüggen, K.-H. Chen, and J.-J. Chen, "End-to-end timing analysis of sporadic cause-effect chains in distributed systems," *ACM Transactions on Embedded Computing Systems*, vol. 18, no. 5s, pp. 1–24, 2019.

[3] F. Siron, D. Potop-Butucaru, R. de Simone, D. Chabrol, and A. Methni, "Formal semantics of the psyc language," Inria Sophia-Antipolis, Tech. Rep., 2023.

[4] T. A. Henzinger, B. Horowitz, and C. M. Kirsch, "Giotto: a time-triggered language for embedded programming," *Proceedings of the IEEE*, vol. 91, no. 1, pp. 84–99, 2003.

[5] E. A. Lee and M. Lohstroh, "Generalizing logical execution time," in *Principles of Systems Design*, ser. Lecture Notes in Computer Science. Cham: Springer Nature Switzerland, 2022, vol. 13660, pp. 160–181.

[6] S. A. Issolah, M. Boyer, D. Chabrol, and G. Phavorin, "Comparison of logical execution time programming languages," in *13th European Congress of Embedded Real Time Systems (ERTS)*, February 2026.

[7] D. Chabrol, "Étude, conception et mise en œuvre d'un protocole de communication synchrone tolérant aux fautes et prédictible sur des composants réseaux standards," Ph.D. dissertation, Université Paris 11, 2006.

[8] D. Chabrol, J. Guyomarc'h, F. Siron, G. Phavorin, S. Thompson, E. Jenn, and F. Thurieau, "Separation of functional and time interference concerns for efficient amc 20-193 compliance," in *ERTS 2024 – Embedded Real Time Systems*, 2024.

[9] E. Farcas, C. Farcas, W. Pree, and J. Templ, "Transparent distribution of real-time components based on logical execution time," in *LCTES*, 2005, pp. 31–39.

[10] K.-B. Gemlau, L. Köhler, R. Ernst, and S. Quinton, "System-level logical execution time: Augmenting the logical execution time paradigm for distributed real-time automotive software," *ACM Transactions on Cyber-Physical Systems*, vol. 5, no. 2, pp. 1–27, 2021.

[11] A. Bouillard, M. Boyer, and E. Le Corronc, *Deterministic network calculus: From theory to practical implementation*. John Wiley & Sons, 2018.

[12] L. Köhler, P. Hertha, M. Beckert, A. Bendrick, and R. Ernst, "Robust cause-effect chains with bounded execution time and system-level logical execution time," *ACM Transactions on Embedded Computing Systems*, vol. 22, no. 3, pp. 1–28, 2023.

[13] W. Baron, A. Arestova, C. Sippl, K.-S. Hielscher, and R. German, "LETT: An execution model for distributed real-time systems," in *IEEE 94th Vehicular Technology Conference (VTC2021-Fall)*. Norman, OK, USA: IEEE, 2021, pp. 1–7.

[14] F. Siron, "Méthodologie de vérification formelle de propriétés temporelles pour les applications temps-réel critiques basées sur le concept de temps logique," Ph.D. dissertation, Sorbonne Université, 2023.

[15] D. Chabrol, G. Phavorin, and E. Jenn, "slet for distributed aerospace landing system," in *Design, Automation & Test in Europe Conference (DATE)*. Valencia, Spain: IEEE, 2024, pp. 1–2.

Path Preselection and Warm-Start Strategies for Network-Calculus-Based Routing Optimization

Ouajih MEKNI
CNRS,LORIA
F-54000 Nancy, France
Email: ouajih.mekni@loria.fr

***Abstract*—Time-Sensitive Networking (TSN) requires deterministic guarantees on end-to-end communication delays while efficiently utilizing network resources. Network calculus provides a mathematical framework for computing such guarantees and can be integrated into routing optimization to jointly determine routing configurations and delay bounds. However, incorporating Network calculus formulations into mixed-integer linear programming (MILP) or mixed-integer nonlinear programming (MINLP) routing optimization models remains computationally challenging. In MILP formulations, nonlinear expressions and logical constraints must be linearized, while both MILP and MINLP formulations can result in large optimization models whose solving time increases rapidly as the network size grows. In this paper, we present a heuristic path preselection framework aimed at reducing the computational cost of MILP or MINLP routing optimization. In particular, we propose efficient candidate path generation heuristics together with warm-start initialization strategies for the MILP or MINLP solvers. The proposed methods are evaluated on the Geyer-Bondorf benchmark and an industrial challenge dataset. Experimental results show a significant reduction in optimization time while maintaining high-quality routing solutions and deterministic delay guarantees verified using Saihu.**

I. Introduction

Time-sensitive networking (TSN) and deterministic networking (DetNet) have emerged as key technologies for supporting applications that require strict guarantees on communication latency and reliability, such as industrial automation, avionics, automotive systems, and critical infrastructures. These networks must satisfy stringent end-to-end timing constraints while efficiently utilizing the available network resources.

Network-calculus (NC) [1][2] provides a mathematical framework for deriving deterministic worst-case delay bounds in communication networks and has become one of the main approaches for validating TSN and DetNet architectures before deployment. Several network-calculus analyses have been proposed, including pay multiplexing only once (PMOO), pay bursts only once (PBOO), and TFA++, each providing different trade-offs between computational complexity and analysis accuracy. Although these methods produce deterministic guarantees, their execution time can become significant as the network size and traffic complexity increase due to the non-linear nature of network calculus computations.

In practice, however, the challenge is not limited to verifying an existing routing configuration. A more difficult problem consists in automatically finding routing configurations that satisfy all network constraints while minimizing the resulting worst-case delays. In this context, delay analysis becomes an integral part of the optimization process, making the overall problem considerably more computationally demanding. In this work, we consider a routing optimization approach in which a TFA++-based Network-calculus delay formulation is embedded into a mixed-integer linear programming (MILP) model, allowing routing decisions and worst-case delay estimation to be optimized simultaneously. While such optimization approaches produce high-quality routing solutions, their solving time grows rapidly with the network size and, in particular, with the number of candidate paths considered for each flow. This scalability issue is not specific to the considered MILP formulation and also arises in MINLP-based routing optimization models. The path preselection framework proposed in this paper addresses this challenge by reducing the search space while preserving solution quality.

Our contributions include improved candidate paths generation strategies, heuristic warm-start initialization methods for the MILP solver, and enhancements to the overall optimization workflow. We experimentally evaluate the impact of these techniques on solving time, optimization quality, and the scalability of the proposed framework.

The remainder of this paper is organized as follows. Section II introduces the Related works. Section III presents the network calculus background . Section IV present the MILP or MINLP formulations . Section V evaluates the performance of our heurstics on representative benchmark networks. Finally, Section VI concludes the paper and discusses future research directions.

II. Related Work

Network calculus [1][2] has become one of the main theoretical frameworks for providing deterministic guarantees in time-sensitive and deterministic networks. Given a routed network together with traffic and service models, network calculus computes an upper bound on the worst-case performance metrics such as end-to-end delays and buffer occupancies. Over the years, several deterministic analyses have been proposed to improve the tightness of these bounds while reducing the pessimism introduced by classical compositional approaches.

Among the best-known analyses are pay bursts only once (PBOO)[3] and pay multiplexing only once (PMOO)[4] . Both approaches exploit the structure of feed-forward networks to avoid repeatedly accounting for burstiness and multiplexing effects. PMOO generally provides tighter delay bounds by considering multiplexing only once along the end-to-end path of a flow, whereas PBOO offers a simpler analysis with lower computational complexity. More recently, TFA++ has been proposed as an efficient alternative, providing a favorable

trade-off between computational cost and analysis accuracy.

Saihu [5] is a software framework that implements these network calculus analyses is called . This tool allows users to evaluate the worst-case performance of TSN and DetNet networks by computing deterministic delay guarantees for a given routing configuration. Its objective is therefore to validate an existing network before deployment, ensuring that the routing satisfies the required timing constraints.

The framework considered in this paper addresses a fundamentally different problem. Instead of analyzing a given routing, the delay formulation is embedded directly into an optimization model, allowing routing decisions and worst-case delay estimation to be optimized simultaneously.

DiffNC [6] formulates network calculus-based routing optimization as a MINLP model by exploiting the differentiability of network calculus delay bounds. Given a predefined set of candidate paths for each flow, it optimizes the routing configuration while satisfying deterministic delay constraints. However, despite the efficiency of gradient-based optimization, the optimization problem remains computationally expensive as the number of candidate paths increases. Consequently, improving the efficiency of this optimization process becomes a key research objective and motivates the acceleration techniques proposed in this work.

III. Network Calculus Background

A. Network Calculus Basics

Network calculus (NC) is a mathematical framework for deriving deterministic worst-case performance guarantees in communication networks. It models both traffic and network resources using curves, allowing upper bounds on metrics such as end-to-end delay and backlog to be computed.

A traffic flow is characterized by an *arrival curve* $\alpha(t)$, which bounds the maximum amount of data that can arrive during any interval of duration t. Throughout this work, flows are modeled by using leaky-bucket arrival curves,

$$\alpha(t) = \begin{cases} rt + b, & t > 0, \\ 0, & t = 0. \end{cases} \tag{1}$$

where r denotes the transmission rate and b the burst size.

Similarly, each network element is modeled by a *service curve* $\beta(t)$ that describes the minimum service guaranteed by the node. A common model is the rate-latency service curve,

$$\beta(t) = R[t - T]^+. \tag{2}$$

where R is the guaranteed service rate, T is the processing latency and $[x]^+ = max(x, 0)$

Given an arrival curve and a service curve, Network calculus derives as shown in the Figure 1 an upper bound on the delay, denoted by D^*, as the maximum horizontal distance between the two curves, and an upper bound on the backlog, denoted by B^*, as the maximum vertical distance. These deterministic guarantees make Network Calculus particularly suitable for the analysis of TSN and DetNet networks.

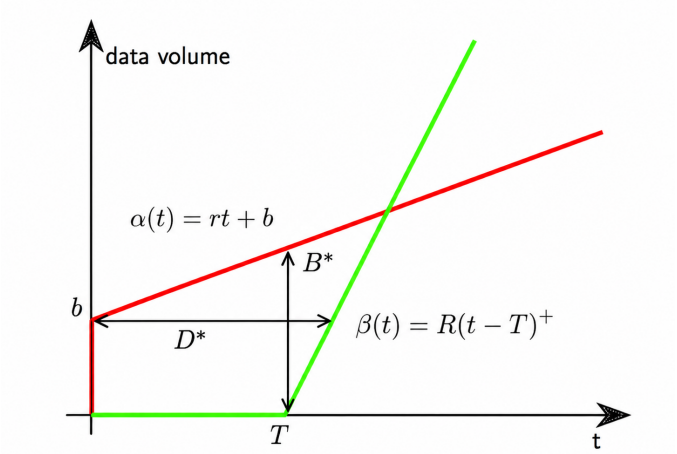


Fig. 1: Network calculus curve. Arrival and service curves that illustrate delay and backlog bounds.

B. Total Flow Analysis (TFA)

Total flow analysis (TFA) is one of the most widely used compositional approaches for FIFO networks based on network calculus. Instead of analyzing the network globally, TFA computes a delay bound independently for each output port. At every node, the arrival curves of all incoming flows are aggregated and compared with the service curve of the node to obtain a local delay bound. Arrival curves are then worsened and propagated to downstream nodes.

The end-to-end delay of a flow is then obtained by summing the delay bounds of every node along its path. This approach has a linear complexity with respect to the network size, making it highly scalable. However, because the worst-case delay is accumulated hop-by-hop, TFA is known to produce pessimistic end-to-end delay bounds.

C. TFA++

TFA++ extends the classical TFA approach by exploiting the line-shaping effect introduced by transmission links. Instead of considering only the aggregated arrival traffic at each node, TFA++ incorporates additional shaping constraints derived from the capacity of incoming links and the maximum packet sizes of the corresponding traffic aggregates.

This additional information produces tighter arrival curves before aggregation, resulting in significantly smaller delay bounds while preserving the linear computational complexity of TFA. Consequently, TFA++ provides an excellent trade-off between analysis accuracy and execution time, making it particularly attractive for deterministic network analysis.

The routing optimization framework considered in this work builds upon the principles of TFA++. In particular, it embeds a simplified TFA++-based delay formulation into a Mixed Integer Linear Programming (MILP) model, allowing routing decisions and delay estimation to be optimized simultaneously.

D. Delay Verification

Although the MILP optimizes the delay bounds predicted by the embedded TFA++ formulation, the obtained routing configurations are independently verified using the Saihu [5]

framework. Saihu implements several network calculus analyses and computes deterministic end-to-end delay bounds for routed networks. This independent verification allows us to evaluate the quality of the optimized routing and compare the MILP-estimated delays with complete network calculus analyses such as xTFA [7] .

IV. Routing Optimization Framework

This section introduces the routing optimization framework considered in this work, followed by the proposed candidate path generation and warm-start techniques.

A. MILP and MINLP Formulation

Given a set of candidate paths for each flow, the routing problem can be formulated either as a MILP model or as a MINLP model. In both cases, a common approach is to associate a binary decision variable $p_{f,i}$ with each candidate path $P_{f,i}$ of flow f, where

$$p_{f,i} = \begin{cases} 1, & \text{if flow } f \text{ is assigned to } P_{f,i}, \\ 0, & \text{otherwise.} \end{cases}$$

Each flow must be assigned to exactly one candidate path,

$$\sum_{P_{f,i} \in P(f)} p_{f,i} = 1, \qquad \forall f \in F. \tag{3}$$

The objective is to minimize the total predicted end-to-end delay of all flows,

$$\min \sum_{f \in F} D_f^{ETE}. \tag{4}$$

subject to routing consistency, link-capacity, and deadline constraints. The resulting optimization problems jointly optimize the routing configuration while guaranteeing deterministic delay bounds for all flows.

B. Candidate Path Generation

The complexity of the path-based problems strongly depends on the number of candidate paths generated for each flow. Enumerating all simple paths rapidly becomes intractable on large network topologies, while providing too few candidate paths may significantly reduce the quality of the optimized routing. To address this trade-off, we develop and evaluate two heuristics path-generation strategies.

The first heuristic referred to as *Classic* computes the Top-k candidate paths using a best-first search guided by a Network Calculus cost function instead of a classical shortest-path metric. Starting from the source node, each partial path is associated with a state composed of its accumulated delay estimate, current burst size, and traversed nodes. When extending a path through an output port with rate R and latency T, the delay estimate and burst are updated according to the TFA recurrence

$$D \leftarrow T + \frac{b}{R}, \qquad b \leftarrow b + rT. \tag{5}$$

where r denotes the flow rate and b its current burst size. The accumulated delay serves as the priority metric during the search, so that paths expected to produce lower deterministic delays are explored first.

To further reduce the search space, only the best k states are maintained for each intermediate node. Additional pruning removes redundant paths and discards candidate states whose estimated cost exceeds a predefined multiple of the current best cost. Consequently, the algorithm generates a small set of high-quality candidate paths while avoiding the exponential complexity of exhaustive path enumeration.

The second heuristic referred to as *Seq-greedy* extends the previous approach by incorporating an approximation of cross-traffic during path generation. Instead of evaluating each flow independently using the nominal service parameters of the network, flows are processed sequentially. After computing the Top-k paths for a given flow, the best candidate is temporarily assumed to be selected and its bandwidth reservation is propagated to the traversed output ports. The effective service rate and latency of these ports are then updated before processing the next flow. Initially, the effective service parameters of each output port are set equal to their original service parameters,

$$R_{\text{eff}} \leftarrow R, \qquad T_{\text{eff}} \leftarrow T.$$

For each flow, candidate paths are evaluated using the current effective service curves. At each traversed output port v, the local delay contribution is estimated as

$$D = T_{\text{eff}}(v) + \frac{b}{R_{\text{eff}}(v)}, \tag{6}$$

and the propagated burst is updated according to

$$b \leftarrow b + rT_{\text{eff}}(v). \tag{7}$$

After selecting the lowest-cost path, the effective parameters of every traversed output port are updated as

$$R_{\text{eff}} \leftarrow R_{\text{eff}} - r, \tag{8}$$

$$T_{\text{eff}} \leftarrow T_{\text{eff}} + \frac{b}{R}. \tag{9}$$

where r and b denote the rate and burst of the selected flow, respectively. These updated effective service parameters are then used to evaluate candidate paths for subsequent flows. Consequently, subsequent flows are evaluated on a progressively loaded network, producing candidate paths that better reflect the expected operating conditions. Although this sequential greedy strategy does not guarantee a globally optimal path generation, it provides a more realistic approximation of resource contention.

C. Warm-start

To further accelerate the optimization process, we initialize the solver with feasible routing solutions obtained from heuristic algorithms. Instead of starting from an empty solution, the solver begins from a routing configuration that already satisfies the routing constraints, providing an initial feasible solution allows the branch-and-bound procedure to converge more rapidly toward high-quality solutions, especially for large network instances.

V. Experimental Evaluation And Results

The proposed optimization techniques were evaluated on two representative benchmark families. The first consists of the Geyer-Bondorf benchmark suite [6] , which is widely used in the network-calculus community to evaluate deterministic routing and delay analysis methods. The second benchmark is an industrial challenge dataset proposed by [8], representing a

realistic TSN network with a significantly larger topology and traffic demand.

A. Geyer-Bondorf Benchmark

The first series of experiments evaluates the proposed path generation and warm-start heuristics on the Geyer-Bondorf benchmark. For each network instance, we compare the optimization execution time and the number of successfully solved MILP instances. In addition, the routing configurations obtained by the MILP are independently verified using Saihu to validate the computed worst-case delay bounds.

TABLE I: Results on the Geyer-Bondorf benchmark.

Method	Time (s)	Number of successful solves
Enumerate all paths	7490	372/381
Classic	295	330/381
Seq-Greedy	314	344/381

The obtained results in the Table I show that reducing the candidate search space together with warm-start initialization significantly decreases the optimization time.The Seq-Greedy approach achieves a higher number of successful solves than the Classic approach, although its overall optimization time is slightly higher. Compared to the approach that considers all candidate paths, the proposed framework achieves a significant reduction in optimization time at the cost of a small decrease in the number of successful solves.

B. Industrial Challenge

To evaluate the scalability of the proposed framework, we additionally considered the industrial challenge network [8] . This benchmark contains a substantially larger topology and traffic set, making the routing optimization considerably more challenging than the synthetic benchmark instances.

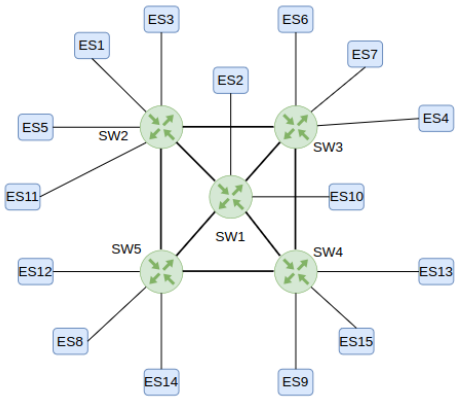


Fig. 2: Industrial challenge topology.

The original network shown in Figure 2 description was first converted into the graph representation required by the optimization framework. In particular, virtual server nodes were introduced to explicitly model the transmission links between output ports and switches, allowing the topology to comply with the representation used by the optimization framework. Candidate paths were then generated using the proposed heuristics before solving the MILP. Finally, the resulting routing configuration was exported and independently verified using saihu.

TABLE II: Results on the industrial challenge benchmark.

Method	Time (s)	Delay-gap compared to enumerate all
Enumerate all paths	223	reference
Classic	50	6%
Seq-greedy	70	-7.9%

The experiments, as shown in Table II, demonstrate that the proposed improvements reduce the optimization effort while maintaining competitive delay guarantees on realistic industrial networks. Table II uses the enumerate all paths method as reference , and compare the gap of delay found by the other two algorithms , To prevent long execution times, all methods were evaluated with a maximum optimality gap of 40% with respect to the LP-relaxed solution, in which the binary decision variables are relaxed to continuous variables. Therefore, the optimization may terminate once the gap between the best feasible solution and the LP-based bound falls below 40%, without reaching the optimal solution. This explain that the seq-greedy method with its negative gap outperformed the enumerate all .

VI. Conclusion

This paper addressed the computational challenges of network calculus-based MILP and MINLP routing optimization, where integrating delay analysis into the optimization process leads to computationally expensive optimization problems.

To address this challenge, we proposed a candidate path generation heuristics step and warm-start initialization strategies to reduce the problem solving time while preserving the quality of the resulting routing solutions. Experimental results on two different datasets demonstrated that the proposed techniques significantly accelerate the optimization process without compromising the deterministic delay guarantees verified using Saihu.

Future work will investigate additional optimization techniques, more advanced path generation heuristics, and learning-based methods for predicting high-quality routing configurations, with the objective of further improving the scalability of routing optimization problems.

References

[1] J.-Y. L. Boudec and P. Thiran, *Network Calculus*. Springer, 2001.

[2] A. Bouillard, M. Boyer, and E. L. Corronc, *Deterministic Network Calculus: From Theory to Practical Implementation*. Wiley, 2018.

[3] M. Fidler, "Extending the network calculus pay bursts only once principle to aggregate scheduling," in *Quality of Service in Multiservice IP Networks*, 2003.

[4] A. Bouillard, P. Nikolaus, and J. Schmitt, "Unleashing the power of paying multiplexing only once in stochastic network calculus," *CoRR*, vol. abs/2104.14215, 2021.

[5] C.-T. Tsai, S. M. Tabatabaee, S. Plassart, and J.-Y. L. Boudec. Saihu: A common interface of worst-case delay analysis tools for time-sensitive networks. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S2352711024002528

[6] F. Geyer and S. Bondorf, "The power of network calculus differentiability," in *IEEE INFOCOM*, 2022.

[7] L. Thomas, "Analysis of the side-effects on latency bounds of combinations of scheduling, redundancy and synchronization mechanisms in time-sensitive networks," Ph.D. dissertation, ISAE-SUPAERO, 2022. [Online]. Available: http://www.theses.fr/2022ESAE0041

[8] M. Boyer and R. Henia, "Embedded reconfiguration of tsn (industrial challenge description)," in *37th Euromicro Conference on Real-Time Systems (ECRTS)*, ser. LIPIcs, vol. 335, 2025, pp. 22:1–22:11.

Automatic Configuration Generation for Real-Time Ethernet Networks

Zakarya HALABI
Sorbonne University, Paris, FRANCE
LIAS, ISAE-ENSMA, Poitiers, FRANCE
Email: zakarya.halabi@ensma.fr

Damien GUIDOLIN--PINA
LyRIDS, ECE Engineering School,
OMNES Education, Paris, France
Email: damien.guidolinpina@gmail.com

Frédéric RIDOUARD
LIAS, ISAE-ENSMA,
Poitiers, FRANCE
Email: frederic.ridouard@ensma.fr

***Abstract*—Ethernet is widely used to implement communication networks in safety-critical systems, where messages are subject to stringent timing constraints. To guarantee compliance with these real-time requirements, various timing analysis techniques, such as Network Calculus, have been proposed. Evaluating these techniques requires common, representative, and reproducible benchmark scenarios. However, existing studies often rely on manually designed examples, making comparisons between approaches difficult and potentially biased.**

In this paper, we present a configurable benchmark generator for real-time Ethernet networks, implemented in Python. From a small set of input parameters, the generator automatically produces complete network configurations, including the topology, traffic flows, routing, and consistent traffic characteristics. The framework provides a simple and reproducible means of generating shared benchmark scenarios, enabling researchers to evaluate, compare, and stress-test timing analysis techniques on identical network configurations. It thereby improves the fairness, repeatability, and reproducibility of experimental evaluations.

I. Introduction

Real-time Ethernet networks are used in critical embedded systems, such as Avionics Full Duplex Switched Ethernet (AFDX) [1] in avionics or Audio Video Bridging (AVB) [2] in the automotive industry. The validation of such systems is not limited to functional correctness but also encompasses temporal correctness.

To verify that such networks satisfy their timing requirements, several evaluation approaches have been developed, including simulation, formal verification, and worst-case timing analysis. Formal verification relies on precise models, such as timed automata and timed Petri nets, whereas worst-case timing analysis includes techniques such as Network Calculus (NC) [3], [4] and Forward end-to-end delay Analysis (FA) [5]. Widely used simulation frameworks include OMNeT++ [6] and ns-3 [7]. These approaches support a broad range of real-time Ethernet technologies, from basic FIFO scheduling to widely deployed standards such as AFDX and more recent mechanisms defined by Time-Sensitive Networking (TSN) [8].

Comparing these diverse and heterogeneous approaches is challenging. Existing studies often rely on manually designed examples, limited configurations, or scenarios tailored to a specific method. This makes fair comparisons difficult and prevents a clear assessment of the conservatism of each approach. This limitation stems from the lack of representative configurations and dedicated tools for their automated generation.

Shared and reproducible benchmarks require complete network configurations, including the network topology (switches, input/output ports, and physical links) and traffic flows with their characteristics, including their paths. Most existing tools focus on a single aspect of network configuration, such as traffic generation or topology generation. To the best of our knowledge, no existing tool automatically generates the topology, traffic flows, and routing required for real-time Ethernet networks such as AFDX within a single reproducible framework.

To address this gap, we propose an open-source configuration generator for real-time Ethernet networks. The tool automatically generates network topologies, traffic flows, routing, and diverse traffic loads in a reproducible manner. It supports several topologies and service policies. The generated scenarios are stored in a structured format that can be used to evaluate and validate timing analysis methods, assess their conservatism, compare different approaches, and identify counterexamples where an analysis fails.

The paper is organized as follows. Section II presents the related work. Section III describes the proposed generator, following the order in which a scenario is constructed. Section IV presents the implementation of the tool. Finally, Section V concludes the paper and discusses planned extensions and future work.

II. Related works

Evaluating timing analysis methods for real-time networks requires representative benchmarks that combine three key elements: traffic models with realistic conditions, diverse and representative network topologies, and well-defined service policies. The following review highlights the strengths and limitations of existing approaches and motivates the design choices of the generator proposed in this paper. We review existing open-source tools for traffic generation, topology generation, and routing, as their source code can be freely inspected, reused, and extended for benchmarking purposes.

Traffic generation: In [9], the authors argue that traffic generators should reflect real-world traffic profiles rather than rely on simple random flows. They propose Swing [10], a closed-loop traffic generator capable of reproducing realistic traffic patterns and their evolution over time. However, Swing targets conventional networks and does not address the specific requirements of real-time Ethernet, such as bounded latency,

periodic traffic, and strict timing constraints. Moreover, it is not integrated with topology or routing generation.

Topology generation: Quoitin et al. [11] propose IGen, a tool for generating Internet topologies at the router level based on network design rules rather than purely random generation. In [12], the authors present TopologyGenerator, which generates dynamic topologies that can evolve over time, while [13] introduces a flexible tool for creating and modifying various topology types for simulation purposes. These tools demonstrate the benefits of automated topology generation for reproducible research. However, they target Internet-scale or simulation-oriented networks and do not capture the specific structure of real-time Ethernet networks, which are typically small-scale, provide predictable and bounded communication delays, and consist of end systems interconnected through a limited number of switches.

Routing and configuration of real-time flows: Determining frame routes is a well-established problem in graph theory. For example, the Breadth-First Search (BFS) algorithm [14] has long been used to compute shortest paths in unweighted graphs and remains a standard building block in generic network routing tools. However, generic shortest-path algorithms do not, by themselves, account for the specific constraints of real-time Ethernet networks, such as flow admission constraints, bounded BAG values, or per-port interference. These constraints must therefore be considered in addition to the routing decision. In [15], Al Sheikh et al. address the problem of configuring virtual links and computing routes in an AFDX network with the objective of minimizing the maximum link utilization. However, their approach assumes that both the network topology and the set of data flows are given beforehand. Moreover, it neither generates new topologies or traffic patterns nor provides a reproducible method for constructing new test cases from scratch.

Overall, these studies address traffic generation, topology generation, and routing as separate aspects. To the best of our knowledge, no existing open-source tool combines these three aspects into a single flexible framework dedicated to real-time Ethernet networks. Our generator addresses this gap by jointly controlling the network topology, traffic parameters, and network load within a single reproducible generation process. The resulting configurations are exported in a structured format that can be readily integrated with existing timing analysis tools.

III. Network configuration generator

The generator starts from a small set of user-defined parameters and automatically produces a complete real-time Ethernet network configuration. As illustrated in Fig. 1, the generation process is performed step by step. First, the switches are created, followed by the end systems (input/output points). The generator then constructs the traffic by successively generating the flows, computing their routes, and assigning their parameters while ensuring that no link is overloaded. Finally, it outputs a structured file containing the complete network configuration. Since each step is deterministic, the generation process is fully reproducible. By default, and in the absence of additional timing constraints such as deadlines, all generated configurations are schedulable, meaning that the end-to-end delay of every frame is bounded.

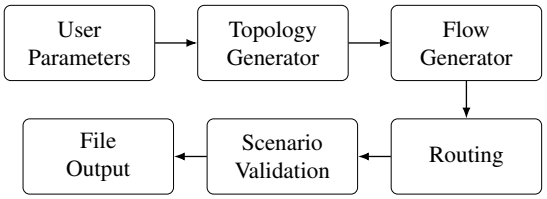


Fig. 1: Main pipeline of the generator.

A. Network Model and Notations

First, we introduce the network model used throughout this paper. End Systems (ESs) represent the ingress and egress points of the network. They are interconnected through a set of switches and physical communication links. The ESs exchange data through mono-emitter, unicast, and sporadic flows. A flow f_i follows a static route from its source to its destination ES and is constrained by a traffic contract at its source ES:

- L_i: the maximum frame size (in bits) generated by f_i;
- BAG_i: the minimum interval between the generation times of two consecutive frames of flow f_i at its source ES.

A switch (Sw) consists of a set of input ports and a set of output ports. It receives frames through its input ports and forwards them to the appropriate output ports according to the flow specifications. Traffic policing and routing are applied at the output ports. Each ES transmits frames through a single output port. The service rate of an output port p, either on a Sw or an ES, is denoted by R^p.

A configuration generated by our framework consists of N_{es} ($N_{es} > 0$) End Systems interconnected by N_{sw} ($N_{sw} > 0$) switches and exchanging data through N_{fl} ($N_{fl} > 0$) flows. First, the user selects a topology (*cf.* Section III-B) and specifies the number of switches N_{sw} (which is one by default), subject to the constraints imposed by the selected topology. The user can then specify the number of End Systems, which defaults to $3 \times N_{sw}$, and the number of flows, which defaults to $N_{fl} = 3 \times N_{sw}$.

The user can also define the frame size bounds $[L_{\min}, L_{\max}]$ and the BAG interval $[BAG_{\min}, BAG_{\max}]$ for all flows. The constant service rate of all output ports, R port, can also be specified and defaults to 100 Mbps. Finally, the service policy applied to each switch output port can be selected as either FIFO or FP/FIFO, the latter enabling priority-based traffic scheduling.

B. Topology generation

The generator supports the following network topologies.

Single node: All End Systems are connected to a single switch. One ES acts as the destination, while all others act as sources. This is the simplest topology and is used as a baseline.

Line 1I/1O: The topology consists of two ESs. The first acts exclusively as a source and is connected to the destination ES through a linear chain of switches.

Line NI/1O: This topology extends the Line 1I/1O topology by distributing multiple source ESs across the switches along the line. The destination ES is connected to the last switch.

Line (NI/NO): This is the general line topology, with multiple source and destination ESs distributed across the switches along the line as illustrated in Fig. 2. Traffic flows in a single direction, resulting in different path lengths and contention points.

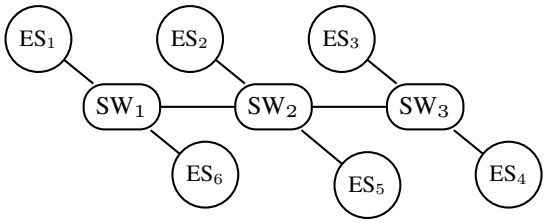


Fig. 2: Line NI/NO topology.

Tree: The switches form a binary tree. Source ESs are connected only to leaf switches, while the destination ES is connected to the root, as illustrated in Fig. 3. Traffic flows toward the root, where flows from different branches may contend. Flows therefore share resources only when they reach a common parent switch.

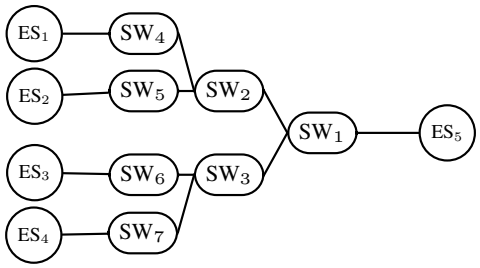


Fig. 3: Tree topology.

Ring: The switches form a directed ring. Source and destination ESs are distributed around the ring, and traffic flows in a single direction. This topology can introduce **cyclic dependencies** between flows, making it particularly suitable for evaluating real-time timing analysis methods.

Random: The switches are interconnected according to a randomly generated graph rather than a predefined structure. Physical communication links are generated randomly while ensuring graph connectivity, so that every switch can reach every other switch. End Systems are then uniformly distributed among the switches. Unlike the structured topologies presented previously, this topology produces heterogeneous scenarios with irregular path lengths and potentially asymmetric traffic patterns, making it suitable for evaluating timing analysis methods under diverse network conditions.

Classical random graph models, such as [16], [17] and [18], either do not guarantee connectivity or introduce specific structural properties, such as hub nodes or distance-dependent link distributions. To avoid these limitations, the generator currently constructs a random spanning tree using a Prüfer sequence [19], which guarantees connectivity and provides uniform sampling of labeled trees. This step is modular and can be replaced by other spanning-tree generation algorithms in future versions. Additional links are then added randomly, producing graphs ranging from sparse trees to highly connected networks without imposing a specific topology, as illustrated in Fig. 4.

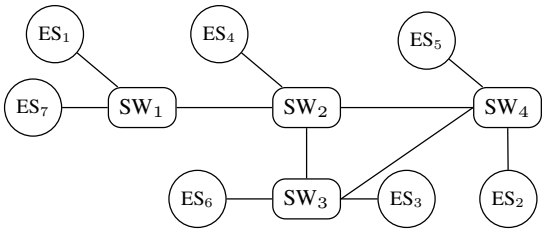


Fig. 4: Random topology.

C. Flow Generation

To generate exactly the user-defined number of flows (N_{fl}), our algorithm assigns the following attributes to each flow:

1) **Source and destination ESs:** source and destination ESs are selected among End Systems that have not yet been used, as long as at least one such ES remains. This ensures that all N_{es} End Systems specified by the user are involved in at least one flow. For the remaining flows, source-destination pairs are randomly generated while respecting the constraints imposed by the selected topology;
2) **Route:** an ordered list of switches connecting the source and destination ESs (*cf.* Section III-C1);
3) **Flow parameters:** the frame size, BAG, and priority are assigned only after all flow routes have been determined, ensuring that the resulting configuration satisfies the required constraints (*cf.* Section III-C2).

1) Routing: Once the source-destination pairs have been defined, routes are computed from the network topology before any traffic parameters are assigned. Thus, routing depends only on the network structure and is independent of the generated traffic. For each source-destination pair, an unweighted shortest-path search based on Breadth-First Search (BFS) is performed. This approach is deterministic and reproducible and assigns a single fixed path to each flow, thereby simplifying interference analysis by avoiding traffic-dependent or multipath routing decisions.

2) Configuration validation: Once all routes have been computed, traffic parameters are assigned while ensuring that no output port is overloaded. To construct a valid configuration, output ports are processed in decreasing order of the number of traversing flows, starting with the most constrained ports.

Let p be an output port and let N_{fl}^{p} denote the number of flows crossing p. We denote by f_k the k^{th} flow analyzed on p, with $1 \leq k \leq N_{fl}^{p}$. Its maximum frame size L_k and BAG BAG_k are randomly selected within the user-defined ranges. The bandwidth requirement of f_k is denoted by $\rho_k = \frac{L_k}{BAG_k}$. To ensure that the total load on p remains less than or equal

to its capacity, the parameters assigned to f_k must satisfy the following condition, which is one of the original contributions of this paper:

$$\rho_k \leq R^p - \sum_{i=1}^{k-1} \rho_i - \left(N_{fl}^p - k\right) \frac{L_{\min}}{BAG_{\max}}.$$

This condition ensures that sufficient residual bandwidth remains for the flows that have not yet been analyzed. More specifically:

- $R^p - \sum_{i=1}^{k-1} \rho_i$ denotes the residual bandwidth available on p before assigning the parameters of f_k;
- $N_{fl}^p - k$ denotes the number of remaining flows to be analyzed on output port p after f_k;
- $\frac{L_{\min}}{BAG_{\max}}$ denotes the minimum possible load of each remaining flow.

Therefore, the selected parameters for f_k leave sufficient bandwidth for all subsequent flows, for which at least the minimum possible load can be assigned. By applying this condition successively to all flows, the generator guarantees that the resulting configuration does not overload any output port.

Finally, when the FP/FIFO policy is selected, a priority level is assigned to each flow. The generated priorities are then incorporated into the corresponding output-port configurations.

D. Output Format

The generated configuration is exported to a file containing:

- the network topology, including the sets of End Systems, switches, and physical communication links;
- the set of flows, including their characteristics (frame size, BAG, and optional priority) and routes (source ES, ordered list of traversed switches, and destination ES).

The exported configuration format is currently directly compatible with OMNeT++.

IV. Implementation

A first implementation of the proposed generator is available as an open-source project [20]. The generator is implemented in Python, runs from the command line, and follows the workflow described in Section III. It is configured through the parameters presented in Section III-A. To ensure reproducibility, the random seed can be fixed, allowing identical scenarios to be regenerated. The current implementation exports scenarios as structured JSON files. To validate the generated configurations, the exported scenarios were also simulated in OMNeT++, allowing the generated behavior to be compared with the expected network operation. As an open-source project, the generator can be freely reused, extended, and adapted to build custom benchmark scenarios or support future research.

V. Conclusion

In this paper, we presented an open-source, configurable scenario generator for real-time Ethernet networks that jointly generates network topologies, traffic flows, routing, and traffic parameters in a reproducible manner. We also introduced a first Python implementation of the generator. This implementation supports seven network topology types, two service policies (FIFO and FP/FIFO), and exports scenarios in a JSON format compatible with OMNeT++. It enables systematic, reproducible, and fair comparisons of timing analysis methods.

Future work will extend the generator in several directions. First, additional export formats will improve interoperability with existing simulation and analysis tools. Second, support for multicast flows will be added to better reflect real AFDX traffic patterns while introducing new challenges for routing and load computation. Finally, it would be interesting to extend the generator to support the creation of large-scale scenarios, including more complex TSN-like configurations.

References

[1] Airlines Electronic Engineering Committee, "Aircraft data network, part 7: Avionics full-duplex switched ethernet network," SAE Industry Technologies Consortium, Annapolis, Maryland, Tech. Rep. ARINC Specification 664 Part 7-1, 2009.

[2] 802.1BA, "Ieee standard for local and metropolitan area networks–audio video bridging (avb) systems," IEEE, Tech. Rep., 2011.

[3] F. Frances, C. Fraboul, and J. Grieu, "Using network calculus to optimize the AFDX network," in *Conference ERTS'06*, 2006.

[4] M. Boyer and C. Fraboul, "Tightening end to end delay upper bound for AFDX network calculus with rate latency fifo servers using network calculus," in *2008 IEEE International Workshop on Factory Communication Systems*. IEEE, 2008, pp. 11–20.

[5] G. Kemayo, N. Benammar, F. Ridouard, H. Bauer, and P. Richard, "Improving AFDX end-to-end delays analysis," in *2015 IEEE 20th Conference on Emerging Technologies & Factory Automation (ETFA)*. IEEE, 2015, pp. 1–8.

[6] Q. Yang, H. Lu, and X. Tu, "Simulation and experiment of AFDX network based on omnet++," in *2020 Chinese Automation Congress (CAC)*. IEEE, 2020, pp. 5849–5854.

[7] O. Ozkaya, J. Haxhibeqiri, I. Moerman, and J. Hoebeke, "Simulating and validating openwifi w-tsn in ns-3," in *2024 IEEE 20th International Conference on Factory Communication Systems (WFCS)*, 2024, pp. 1–4.

[8] L. Deng, G. Xie, H. Liu, Y. Han, R. Li, and K. Li, "A survey of real-time ethernet modeling and design methodologies: From AVB to TSN," *ACM Computing Surveys*, vol. 55, no. 2, pp. 1–36, 2022.

[9] K. V. Vishwanath and A. Vahdat, "Realistic and responsive network traffic generation," in *Proceedings of the 2006 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications (SIGCOMM)*. New York, NY, USA: ACM, 2006, pp. 111–122.

[10] ——, "Swing: Realistic and responsive network traffic generation," *IEEE/ACM Transactions on Networking*, vol. 17, no. 3, pp. 712–725, 2009.

[11] B. Quoitin, V. Van den Schrieck, P. François, and O. Bonaventure, "IGen: Generation of Router-Level Internet Topologies through Network Design Heuristics," in *2009 21st International Teletraffic Congress (ITC 21)*. Paris, France: IEEE, 2009, pp. 1–8. [Online]. Available: https://dial.uclouvain.be/pr/boreal/en/object/boreal%3A67564

[12] SpaceNetLab, "TopologyGenerator: A generator for constructing dynamic network topology," https://github.com/SpaceNetLab/TopologyGenerator, 2020, accessed: 2026-06-29.

[13] J. Domke, "TopologyGenerator: Tool to create/modify various types and sizes of network topologies," https://gitlab.com/domke/TopologyGenerator, 2025, accessed: 2026-06-29.

[14] E. F. Moore, “The shortest path through a maze,” in *Proceedings of the International Symposium on the Theory of Switching*. Harvard University Press, 1959, pp. 285–292.

[15] A. Al Sheikh, O. Brun, M. Cheramy, and P.-E. Hladik, “Optimal design of virtual links in AFDX networks,” *Real-Time Systems*, vol. 49, no. 3, pp. 308–336, 2013.

[16] P. Erdős and A. Rényi, “On random graphs,” *Publicationes Mathematicae*, vol. 6, pp. 290–297, 1959.

[17] A.-L. Barabási and R. Albert, “Emergence of scaling in random networks,” *Science*, vol. 286, no. 5439, pp. 509–512, 1999.

[18] B. M. Waxman, “Routing of multipoint connections,” *IEEE Journal on Selected Areas in Communications*, vol. 6, no. 9, pp. 1617–1622, dec 1988.

[19] H. Prüfer, “Neuer beweis eines satzes über permutationen,” *Archiv der Mathematik und Physik*, vol. 27, pp. 742–744, 1918.

[20] Z. Halabi, D. Guidolin-Pina, and F. Ridouard. (2026) An open-source configurable scenario generator for real-time ethernet networks. LIAS Laboratory. Projet de simulation réseau AFDX. [Online]. Available: https://forge.lias-lab.fr/afdx-network-traffic-simulator

Preliminary Analysis of WCTT Pessimism in Meshed Networks with Round-Robin Scheduling and Cyclic Dependencies

Wassim Ben Jabria*†, Jean-Luc Scharbarg*, Jérôme Ermont*, Marc Pantel*, Philippe Baufreton†
* IRIT, Toulouse INP, University of Toulouse, Toulouse, France
{wassim.benjabria, jean-luc.scharbarg, jerome.ermont, marc.pantel}@toulouse-inp.fr
† Safran Electronics & Defense, Paris, France

***Abstract*—Distributed meshed communication architectures can improve fault tolerance by allowing multiple disjoint paths between a source and a destination, but they complicate worst-case traversal time (WCTT) analysis. This paper presents preliminary results on two challenges arising in Network Calculus-based analysis of such architectures. First, Round-Robin scheduling is compared with FIFO; heterogeneous frame sizes and high load can significantly reduce the effective service and lead to pessimistic bounds. Second, cyclic dependencies caused by jitter and burst propagation are resolved using a Gauss-Seidel fixed-point method. Preliminary results show that these cycles can significantly increase WCTT pessimism, especially under high load or strong flow contention, while all tested scanning orders converge to the same fixed point but require different numbers of iterations.**

***Index Terms*—Real-time networks, Network Calculus, Worst-Case Traversal Time (WCTT), Round-Robin Scheduling, Cyclic Dependencies, Fixed-Point Iteration, Avionics Networks**

I. Introduction

Critical real-time embedded communication networks must guarantee high levels of availability, reliability, and temporal determinism, while meeting increasingly stringent integration and certification constraints. Current avionics architectures mainly rely on redundant communication networks organized around acyclic switched topologies, such as those specified by ARINC 664 [1]. These architectures facilitate temporal analysis and certification. This relies on the computation of a safe upper bound on the traversal time of each flow. Several approaches have been proposed for this WCTT analysis [2]–[5]. In this paper, we focus on Network Calculus because it provides formal guarantees on latency and backlog, can be computed efficiently, and has been successfully applied to certification-oriented analyses of avionics communication networks, notably for the evaluation of end-to-end delay bounds in AFDX networks [1] for the Airbus A380.

Nevertheless, AFDX-like architectures offer limited flexibility with respect to architectural evolutions and network failures. These shortcomings could be mitigated with a distributed meshed communication architecture [6]. Redundancy is carried by the network mesh: multiple disjoint routes are assigned between a source and a destination. It improves fault tolerance and reduces dependence on centralized switches.

Figure 1 illustrates a simplified example of the considered architecture with redundant routing. Scheduling of frames at switch output ports can be FIFO or input-port Round-Robin. The latter eliminates the need for a global queue per output port.

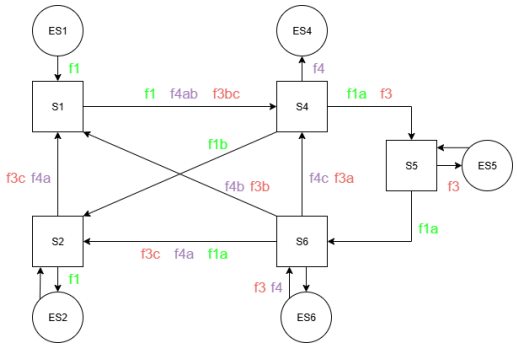


Fig. 1. Meshed Network Architecture with redundant routing.

Network calculus can cope with both FIFO and Round-Robin. It has been shown that the overestimation of worst-case delays is limited with FIFO [7]. This overestimation has not been evaluated for Round-Robin. The first contribution of the paper is to provide such a preliminary evaluation.

In current avionics networks such as AFDX, the routing of flows guarantees acyclic feedforward topologies. Therefore WCTT computation requires a single path: the delay bound in a given port p is computed when bounds on previous ports visited by flows crossing p have been computed. An acyclic topology is very difficult, if not impossible to obtain with the mesh architecture considered in this paper. It comes from the routing of flows on redundant paths on the same mesh architecture. Network calculus can take into account cyclic dependencies thanks to fixed-point methods. These methods increase the computation time and the pessimism of the bounds. The second contribution of the paper is to provide a first evaluation of this increase.

II. WCTT Analysis Using Network Calculus

Network Calculus models traffic using an arrival curve $\alpha(t)$, which upper-bounds the amount of traffic that may arrive, and a service curve $\beta(t)$, which lower-bounds the minimum guaranteed service. The worst-case delay bound is obtained

from the maximum horizontal deviation between these two curves.

A. Impact of Scheduling on WCTT Bounds

The scheduling policy directly influences the guaranteed service. To illustrate this, consider the output link $S1 \rightarrow S4$ in Figure 1. Several flows arriving through different input links compete for the same output.

Under FIFO scheduling, packets are served according to their global arrival order. For example, if they arrive in the order $F1a, F1b, F4a, F4b, F3c, F3b$, they are transmitted in the same order. All competing flows are therefore aggregated into a single queue, and the arrival curve used for the analysis represents the whole traffic sharing $S1 \rightarrow S4$.

Under Round-Robin scheduling, packets are separated according to the input link through which they enter $S1$, and the corresponding queues are served cyclically. With the same set of packets, a possible transmission order becomes $F4a \rightarrow F4b \rightarrow F1a \rightarrow F3c \rightarrow F3b \rightarrow F1b$. The service order therefore depends not only on the global arrival order, but also on the input queue of each flow and on the current state of the Round-Robin scheduler. In the worst case, competing queues may transmit their largest frames before the analyzed queue is served. If the analyzed queue carries a small frame, it may therefore wait for much larger frames to be transmitted first, reducing its minimum guaranteed service and potentially increasing the resulting WCTT bound.

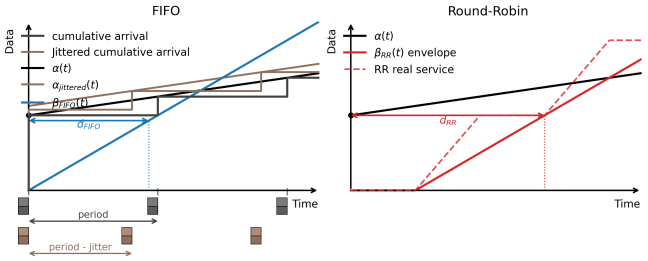


Fig. 2. Network Calculus delay computation under FIFO and Round-Robin, with jitter propagation.

Figure 2 illustrates these mechanisms through the construction of arrival curves, FIFO and Round-Robin service curves, local delay computation, and burst propagation.

Therefore, FIFO and input-port Round-Robin do not lead to the same service behavior and may produce different WCTT bounds for the same set of flows. This motivates the study presented in Section III, which evaluates the impact of Round-Robin scheduling using FIFO as a reference.

B. Jitter Propagation and Cyclic Dependencies

The WCTT of a flow is computed along the successive links of its route. Delay variations experienced by packets of the same flow introduce jitter, whose effect must be propagated from one link to the next.

As illustrated in Figure 2, jitter shifts the initial arrival curve $\alpha(t)$ to obtain $\alpha_{\text{jittered}}(t)$. This shift increases the burst from B to $B' = B + rJ$, where r is the flow rate and J the jitter. The updated burst then defines the new arrival curve used for the next link.

For example, along route $F1a$ in Figure 1, the delay computed on $ES1 \rightarrow S1$ modifies the burst used on $S1 \rightarrow S4$. The same mechanism is then repeated on $S4 \rightarrow S5$, $S5 \rightarrow S6$, $S6 \rightarrow S2$, and $S2 \rightarrow ES2$.

In a feedforward topology, this propagation can be performed sequentially according to an acyclic dependency order: upstream links are analyzed before downstream links, and each computation depends only on already evaluated links.

This property no longer holds in the considered meshed topology. The selected routes may create circular dependencies between delay computations. In the example of Figure 1, the delay computation on $S1 \rightarrow S4$ depends on the burst propagated through $S2 \rightarrow S1$, which depends on $S6 \rightarrow S2$, then on $S5 \rightarrow S6$, $S4 \rightarrow S5$, and finally again on $S1 \rightarrow S4$. The resulting dependency cycle is therefore: $S1 \rightarrow S4 \rightarrow S5 \rightarrow S6 \rightarrow S2 \rightarrow S1$.

This cycle represents a dependency between delay computations [8], not merely a physical path in the network. The WCTT can therefore no longer be computed through a single feedforward pass and must instead be formulated as a fixed-point problem. Starting from initial burst values, delays, jitters, and bursts are iteratively recomputed until convergence within a tolerance ε.

These cyclic dependencies may increase both the pessimism of the computed bounds and the number of iterations required for convergence. Their impact is investigated in Section IV.

III. Impact of Round-Robin Scheduling on WCTT Bounds

Scheduling policies directly affect the service offered to network flows and therefore the resulting worst-case delay bounds. In deterministic network analysis, FIFO scheduling is commonly adopted because of its simplicity and its widespread use in existing worst-case delay analyses. In contrast, although Round-Robin scheduling is required by the target architecture considered in this work, its impact on Network Calculus delay bounds remains less understood. The objective of this study is therefore not merely to compare FIFO and Round-Robin schedulers, but to identify the traffic conditions under which Round-Robin may lead to pessimistic or even unbounded delay bounds. FIFO is used as a reference to better understand the mechanisms responsible for these differences. To isolate the impact of scheduling, we consider the single-node topology shown in Figure 3 and perform a Network Calculus analysis including serialization effects, so that two frames arriving from the same input link cannot be received simultaneously.

Four representative scenarios were initially studied to identify the main sources of pessimism under Round-Robin scheduling. This paper focuses on the two scenarios exhibiting the largest differences with FIFO: frame-size heterogeneity (S2c) and its combination with increasing network load (S4a).

Both scenarios consider a 2500 Mbps output link carrying 16 flows. S2c studies a balanced traffic distribution with het-

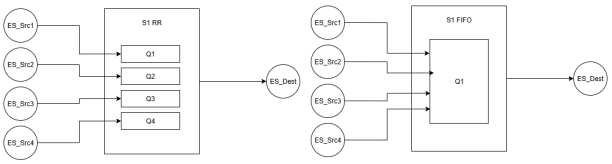


Fig. 3. Network architecture with Round-Robin and FIFO node scheduling.

erogeneous frame sizes, whereas S4a combines heterogeneous frame sizes with increasing network load.

A. S2c – Impact of Heterogeneous Frame Sizes

The representative configuration consists of four flows per input port, with two 1000-bit frames and two 12000-bit frames on each input link. Figure 4 compares the FIFO and Round-Robin service curves. In Round-Robin, the analyzed queue is assumed to be served after all competing queues, which themselves transmit their largest possible frames before the analyzed queue transmits its smallest frame. This worst-case construction drastically reduces the effective service perceived by the analyzed queue.

For this configuration, the effective throughput is approximately 68 Mbps, corresponding to nearly 37 times less than the physical link capacity (2500 Mbps) and nine times less than the ideal Round-Robin fair share (625 Mbps). This illustrates how heterogeneous frame sizes alone can make the Round-Robin service model highly pessimistic.

To quantify this phenomenon, the frame-size ratio was progressively increased while keeping the topology unchanged. The resulting delay bounds are illustrated in Figure 4.

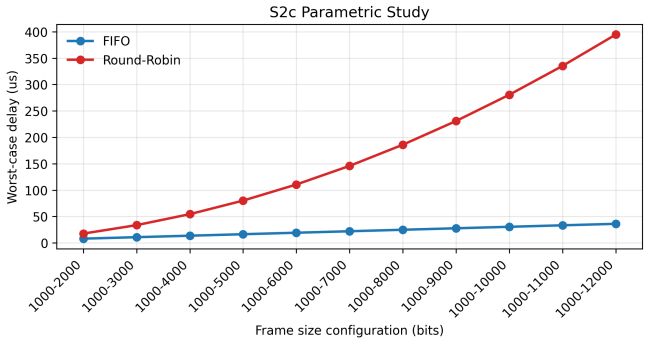


Fig. 4. Evolution of FIFO and Round-Robin delay bounds as a function of frame-size heterogeneity.

Figure 4 shows that FIFO delay increases almost linearly with the maximum frame size, whereas Round-Robin exhibits a pronounced non-linear growth. This behavior is linked to the decrease of the effective throughput perceived by the analyzed queue as the frame-size ratio $\rho = L_{\max}/L_{\min}$ increases. In the tested configurations, this throughput drops from 357 Mbps for a 1000–2000 bit configuration to 68 Mbps for 1000–12000 bit. Consequently, heterogeneous frame sizes sharply reduce the effective Round-Robin service and constitute a major source of pessimism in the resulting delay bounds.

B. S4a – Combined Effect of Heterogeneous Frame Sizes and Network Load

This configuration reveals a critical behavior of Round-Robin scheduling. Queues carrying the smallest frames are the most penalized because they may have to wait for the transmission of larger frames from competing queues before transmitting only a small amount of data. Consequently, their effective service rate decreases and may eventually become lower than the incoming traffic rate, leading to unbounded analytical delay bounds.

To further investigate this phenomenon, the BAG was varied to change the maximum output-link load while keeping the topology and frame sizes unchanged. The resulting delay bounds are shown in Figure 5.

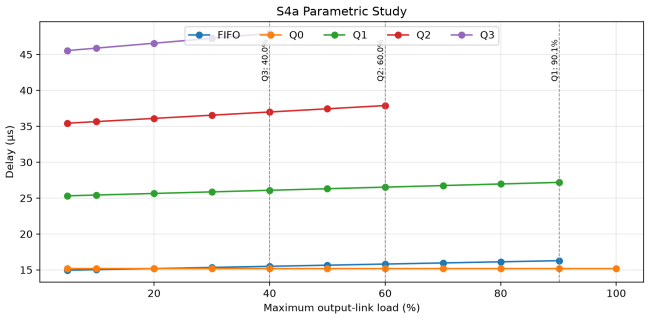


Fig. 5. Delay bounds as a function of maximum output-link load.

Unlike FIFO, which remains bounded over the entire tested load range, Round-Robin exhibits successive instabilities as the maximum output-link load increases. Queues carrying the smallest frames diverge first because they receive the lowest effective service rates.

The theoretical divergence thresholds are obtained by comparing the long-term traffic rate, $12000/\mathrm{BAG}$, with the effective service rate of each queue. They occur at BAG values of 10, 20, 30, and 40 μs for Q0, Q1, Q2, and Q3, respectively, corresponding to network loads of 192%, 96%, 64%, and 48% of the 2500 Mbps link capacity. Therefore, queues carrying the smallest frames become unstable even though the overall network utilization remains well below 100%.

Overall, across the two scenarios, Round-Robin yields delay bounds up to approximately eleven times those of FIFO and can make the smallest-frame queue unbounded at only 48% maximum output-link load.

IV. WCTT Analysis with Cyclic Dependencies – Work in Progress

Cyclic dependencies arise when the delay computed on one link depends on bursts and jitters that themselves depend on other links of the same cycle. In this case, WCTT analysis can no longer be performed through a single feedforward pass and must instead be solved as a fixed-point problem.

To study these dependencies, we consider the topology shown in Figure 6, containing a main cycle and a nested cycle. Dependencies between delay computations are represented using a dependency graph, where vertices correspond to delay equations and edges represent dependencies. Strongly Connected Components (SCCs) [9] identify maximal sets of mutually dependent equations that must be solved jointly.

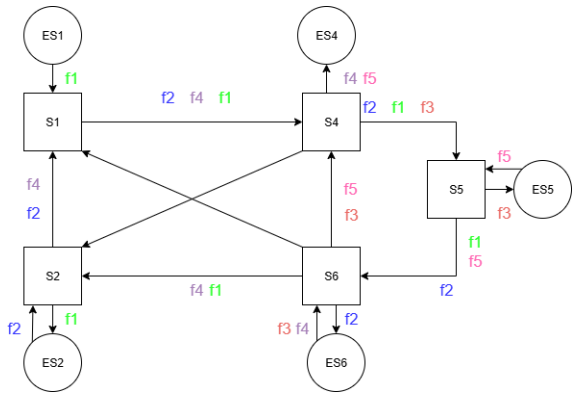


Fig. 6. Example of a topology containing a main cycle and a nested cycle.

Figure 7 shows the dependency graph automatically constructed from the delay equations. In this example, the links S1→S4, S4→S5, S5→S6, S6→S2, S2→S1, and S6→S4 belong to the same SCC and therefore cannot be solved independently.

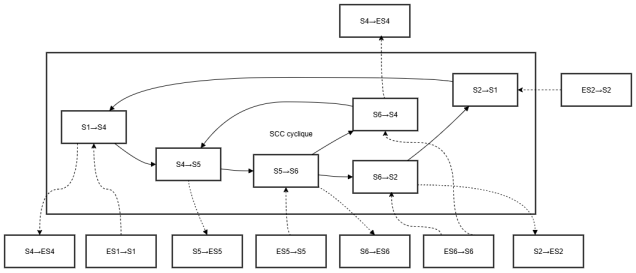


Fig. 7. Dependency graph and strongly connected component.

Each SCC is solved using a Gauss-Seidel fixed-point iteration [10]. Delays and jitters are computed sequentially, and updated bursts are immediately propagated to dependent links. Iteration stops when the maximum burst variation falls below ε.

Five scanning criteria were evaluated in both directions: route order, dependency out-degree, flow count, initial burst sum, and initial rate sum. These criteria capture propagation direction, graph influence, contention, and the terms controlling burst amplification, $B' = B + rJ$. All orders converged to the same fixed point within 10^{-3}. Route order required 5 and 7 iterations for $\varepsilon = 10^{-3}$ and 10^{-9}, respectively, versus 13 and 21 in reverse order; the other strategies required 7–9 and 11–15 iterations. Following the propagation direction therefore reduced the iteration count by up to 67%.

A second preliminary study evaluated the pessimism introduced by cyclic dependencies. Compared with the delay bound before cyclic jitter propagation, the bound of f_1 increases by 5% at $BAG = 100$ μs and by 158% at $BAG = 10$ μs. Similarly, increasing the number of flow copies from 1 to 10 raises this amplification from 5% to 121%. This increase does not result from a change in the local worst-case scenario, but from the iterative propagation of jitter and burstiness within the SCC, which progressively amplifies the arrival curves until convergence.

These preliminary results show that cyclic dependencies do not prevent WCTT computation, but may amplify delay bounds and increase convergence costs. The ongoing study aims to characterize the cycle properties and topology or traffic configurations leading to strong amplification, evaluate update orders that reduce convergence costs, and derive risk indicators computable before convergence.

V. Conclusion

This paper investigated two sources of WCTT pessimism. In our tests, Round-Robin pessimism increased with frame-size heterogeneity and load, reaching eleven times the FIFO bound and making the smallest-frame queue unbounded at 48% load. For cycles, route order was the most efficient strategy, while all orders reached the same fixed point. Cyclic amplification reached 158% under high load and 121% with ten flow copies. Round-Robin and cyclic dependencies therefore add configuration and routing constraints to limit pessimism: homogeneous frame sizes, balanced link loads, and moderate utilization. Future work will validate these trends on an industrial avionics case study and identify risky configurations.

References

[1] "ARINC specification 664: Aircraft data network, parts 1, 2, and 7," Aeronautical Radio, Inc., Tech. Rep., 2002.

[2] J.-Y. Le Boudec and P. Thiran, *Network Calculus: A Theory of Deterministic Queuing Systems for the Internet*. Springer, 2001.

[3] N. Benammar, F. Ridouard, H. Bauer, and P. Richard, "Forward end-to-end delay analysis for AFDX networks," *IEEE Transactions on Industrial Informatics*, 2017.

[4] X. Li, O. Cros, and L. George, "The trajectory approach for AFDX FIFO networks revisited and corrected," in *Proceedings of the IEEE International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA)*, 2014.

[5] A. Soni, J.-L. Scharbarg, and J. Ermont, "A hybrid approach to WCTT analysis in a real-time switched ethernet network," in *Proceedings of the IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2024.

[6] F. Champenois, F. Brandner, T. Grandpierre, E. Borde, A. Suissa, and L. Georges, "Multi-criteria optimization of distributed real-time network topologies," in *Proceedings of the IEEE International Symposium on Real-Time Distributed Computing (ISORC)*, 2024.

[7] H. Bauer, J.-L. Scharbarg, and C. Fraboul, "Improving the worst-case delay analysis of an AFDX network using an optimized trajectory approach," *IEEE Trans. Ind. Informat.*, 2010.

[8] A. Finzi and S. S. Craciunas, "Breaking vs. solving: Analysis and routing of real-time networks with cyclic dependencies using network calculus," in *Proceedings of the International Conference on Real-Time Networks and Systems (RTNS)*, 2019.

[9] R. Tarjan, "Depth-first search and linear graph algorithms," *SIAM Journal on Computing*, 1972.

[10] A. J. Salgado and S. M. Wise, *Classical Numerical Analysis*. Cambridge University Press, 2022.