Deep Q-learning pour la gestion de réseaux déterministes

Adrien Roberty
et Siwar Ben Hadj Said
Université Paris-Saclay,
CEA, List,
F-91191, Palaiseau, France
Email : {adrien.roberty, siwar.benhadjsaid}@cea.fr

Frederic Ridouard,
Henri Bauer
et Annie Geniet
LIAS-Université de Poitiers et ISAE-ENSMA
Futuroscope Cedex, France,
Email : {frederic.ridouard, henri.bauer, annie.geniet}@ensma.fr

***Résumé*—L'industrie 4.0 implique la mise en réseau des équipements de production grâce à l'ensemble de normes réseau Time-Sensitive Networking (TSN) basé sur Ethernet commuté. Ces mécanismes sont configurables dynamiquement. Cet article présente une première architecture permettant la configuration dynamique sous TSN en utilisant l'intelligence artificielle.**

I. Introduction

Le but de l'industrie 4.0 est de réduire les interventions humaines ainsi que les coûts et la consommation d'énergie au strict minimum, tout en augmentant la productivité. Une de ses caractéristiques principales est la mise en réseau des équipements de production : machines, lignes de production, robots, convoyage et stockage. Ceci permettra aux équipements de production d'être capables de se contrôler, de se configurer et d'échanger des informations entre eux. Cela implique des exigences de fiabilité, de latence et de longévité des périphériques de communication élevées. D'un point de vue réseau, l'objectif le plus important pour l'industrie 4.0 reste la performance en temps réel des communications. Cet objectif peut être atteint grâce au Time Sensitive Networking (TSN), un ensemble de normes visant à ajouter des caractéristiques temps-réel à Ethernet. TSN est un ensemble d'une vingtaine de normes pour la mise en place de transmissions, sensibles au temps, de paquets sur des réseaux Ethernet déterministes. Elles sont édictées par un comité au sein du groupe de travail 802.1 de l'Institute of Electrical and Electronics Engineers (IEEE). On peut les décomposer en 4 parties :

1) la synchronisation du temps ;
2) la fiabilité ;
3) les temps de latences garantis ;
4) la gestion des ressources.

Une usine 4.0 est composée de postes de travail mobiles et flexibles qui peuvent être agencés de manières différentes en fonction de la production. Cette reconfiguration des postes de travail nécessite une reconfiguration du réseau qui pourrait être dynamique. De plus, la combinaison des normes TSN rend la configuration d'un réseau TSN difficile, car les interactions entre ces différentes normes ne sont pas forcément triviales. D'autant qu'il y a toujours un plus grand nombre de configurations possibles que de configurations faisables (qui respectent les contraintes applicatives). Les techniques de l'Intelligence Artificielle peuvent apporter une solution. D'où la question de savoir si l'IA peut aider à trouver dynamiquement la bonne configuration pour TSN. Autrement dit, lors d'un changement de topologie réseau, une IA peut-elle, de façon autonome, adapter la configuration TSN ?

Cet article est construit comme suit : la section II résume l'état de l'art, la section III formalise le problème, la section IV explique la méthodologie et la section V conclut et donne des perspectives.

II. État de l'art

De nombreux travaux ont été menés afin de résoudre le problème de la configuration de TSN. Par exemple, dans [2], on rend « intelligents » chaque commutateur réseau grâce à un agent de configuration intégré. Cet agent surveille le réseau en permanence et, lorsqu'il détecte un changement (via un message réseau), adapte la configuration du commutateur et propage l'information aux autres commutateurs. Il s'agit donc d'un système distribué.

L'IA en particulier a déjà été envisagée comme faisant partie de la solution de la problématique de la configuration de TSN . Dans [1], elle est utilisée pour ordonnancer les flux. Dans [3], les auteurs utilisent une IA afin de déterminer si une configuration possible (donc déjà trouvée) est faisable, c'est-à-dire si elle respecte les critères applicatifs. Pour ce faire, ils testent des algorithmes simples d'apprentissages supervisés et non supervisés afin de classer les configurations possibles en faisables/non faisables.

Les travaux qui se rapprochent le plus de l'objectif que nous cherchons à atteindre ont été menés dans [4]. L'auteur utilise l'apprentissage par renforcement afin d'aider à la configuration de réseaux TSN. Plus précisément, il essaye deux algorithmes, Deep Q-learning et acteur-critique. Cependant, ici l'IA se contente de chercher les latences maximales garanties pour chaque classe de trafic critique. Son but est de trouver les latences qui maximisent le nombre de flux pouvant être transmis sans violer les contraintes. C'est un framework qui s'occupe de configurer et simuler le réseau. De plus, les topologies considérées sont seulement linéaires, dit autrement, il n'y a qu'un seul chemin possible entre deux extrémités du réseau.

III. FORMALISATION DU PROBLÈME

Ce sujet comprend deux volets : une partie Ethernet TSN et une partie Intelligence Artificielle. Pour le moment, des simplifications sont appliquées. À terme, plusieurs normes TSN seront prises en compte, de même, plusieurs algorithmes d'apprentissage seront étudiés.

A. Ethernet TSN

Il y a 2 critères particulièrement importants à respecter dans un réseau déterministe :

1) le délai de transmission (latence) maximum ne doit pas être dépassé ;
2) la variation de délai de transmission de bout en bout entre des paquets d'un même flux (gigue) doit tendre vers 0.

Les articles cités dans [5] fournissent une introduction aux normes TSN en général ainsi que des exemples de cas d'applications plus détaillés. Dans un premier temps, nous allons nous intéresser à la norme IEEE 802.1Qbv, qui traite de l'ordonnancement du trafic afin de garantir les temps de latences.

1) Paramètres de configuration à prendre en compte pour Qbv: La norme IEEE 802.1Qbv - *Amendment 25 : Enhancements for Scheduled Traffic* [6] est un amendement à la norme IEEE 802.1Q. Il s'agit d'un mécanisme, similaire à Time-Division Multiple Access (TDMA), appelé en anglais « Time-Aware Shaper » (TAS). Le temps de transmission est divisé en cycles de durée constante. Ce cycle est lui-même divisé en séquences de temps de durées variables. Puis on définit dans quelles séquences une trame est autorisée à être envoyée en fonction de sa classe de trafic. Quand plusieurs classes de trafic sont transmise en même temps, c'est la priorité de la classe qui détermine l'ordre de transmission. Comme première étape, nous allons considérer 2 classes de trafic : le trafic critique avec une priorité élevée et le trafic « normal » avec une priorité basse. Ceci permet de simplifier la configuration des séquences de temps : il y en a 2. Une séquence de temps est utilisée par le trafic TSN (trafic critique), et l'autre séquence de temps est pour le reste du trafic. Cette seconde séquence de temps peut être divisée en deux parties, si on décide de faire démarrer la séquence TSN après le début du cycle et qu'elle se finisse avant la fin du cycle. Par conséquent, la configuration du réseau TSN comprend 2 paramètres :

1) la durée du cycle ;
2) la date de début de la séquence TSN.

Ces paramètres sont très importants pour TSN : si leurs valeurs ne sont pas bonnes, la latence du trafic critique sera impactée ou la gigue du trafic critique sera trop grande ; dans les deux cas, les exigences en termes de qualité de service seront violées. Dans ce cas, on perd l'intérêt de faire de temps-réel.

Nous considérerons ici que les trames du trafic critique font 500 octets et que les autres font 1500 octets. Afin d'éviter le gaspillage de la bande passante, la durée de la séquence de temps réservée à TSN est égale à 2 fois la durée pour envoyer une trame TSN. Les liens ayant une capacité de 100 Mbit/s, la séquence de temps de TSN dure donc 80μs.

2) Topologie considérée: Nous allons nous limiter dans un premier temps à une topologie fixe, composée d'un commutateur et de deux stations, à savoir un émetteur et un récepteur. Les liens quant à eux ont tous une capacité de 100 Mbit/s. La topologie est montrée sur la figure 1. À terme, bien évidemment, la topologie considérée sera complexe et dynamique.

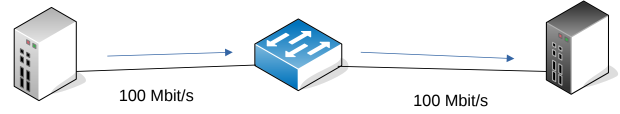

FIGURE 1. La topologie considérée

B. Intelligence artificielle

Il existe un paradigme d'apprentissage adapté à la prise de décision : *l'apprentissage par renforcement*, où un agent interagit avec un environnement. On peut modéliser dans la plupart des cas ceci grâce aux *processus de décisions Markoviens* (MDP) : à chaque instant t, *l'agent* observe un *état* S_t de *l'environnement* et sa *récompense* R_t (obtenu précédemment) et doit prendre une *action* A_t en conséquence. L'état passe alors à S_{t+1} et l'agent reçoit une récompense R_{t+1} (voir figure 2). Une propriété intéressante des MDP est que seul l'état actuel affecte l'état suivant (voir [7] pour plus de détails).

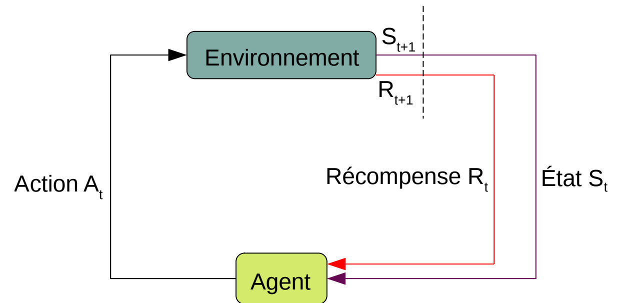

FIGURE 2. Les interactions entre l'agent et l'environnement dans un MDP

1) L'état: L'état du système peut être vu comme une image du système à un instant t à partir duquel l'agent pourra prendre des décisions. Il comprend des informations sur la topologie, les flux, les contraintes à respecter et la configuration TSN. La composition détaillée de l'état est donné dans la table I. À noter que l'état dans lequel les latences et la gigue ont été respectées s'appelle *l'état terminal*.

2) L'action: L'action consiste à modifier une ou plusieurs valeurs de la configuration. Il y a deux actions possibles par valeur à (potentiellement) modifier : incrémenter ou décrémenter. On considère que ne pas modifier une valeur revient à faire $+0$ dessus. Étant donné que la configuration à modifier contient 2 paramètres, il y a donc 4 actions possibles en tout.

TABLE I
L'ÉTAT

Topologie	Nombre de stations Nombre de commutateurs Capacité des liens
Flux	Nombre de flux Longueur des trames TSN Longueur des autres trames
Contraintes	Latence maximum autorisée Nombre de flux rejetés
Configuration TSN	Durée du cycle Date de début de la séquence TSN

3) La récompense: La récompense sert à évaluer la nouvelle configuration. Si la latence ET la gigue sont meilleures que lors de l'état précédent, la récompense sera positive (1). Si la latence OU la gigue est meilleure, la récompense sera nulle (0). Si la latence ET la gigue sont moins bonnes, la récompense sera négative (-1). De cette façon, l'IA cherchera des configurations où la latence tendra de plus en plus vers la latence maximum autorisée jusqu'à devenir inférieure et où la gigue tendra de plus en plus vers 0, jusqu'à atteindre un état terminal.

4) Algorithme d'apprentissage: L'algorithme d'apprentissage retenu dans un premier temps est *Deep Q-learning*. Deep Q-learning a été défini dans [8], [9]. Il s'agit d'un algorithme d'apprentissage par renforcement utilisé pour jouer à un jeu appelé *Atari*. Son algorithme simplifié est présenté dans l'algorithme 1. Q est la fonction de valeur, elle permet de déterminer la récompense potentielle sur le long terme en fonction des actions effectuées. Dans le cas de Deep Q-learning, Q est en fait un réseau de neurones profond. Cela permet d'approximer la récompense à venir lorsque le nombre d'états est potentiellement important. Ne pas faire d'approximation nécessiterait d'enregistrer tous les résultats obtenus grâce à Q pour chaque paire état-action. La politique comportementale est la manière dont sont choisies les actions durant l'apprentissage.

Algorithme 1 : Version simplifiée de Deep Q-learning

1 Initialiser Q;
2 Initialiser état S_0;
3 **tant que** S_t *n'est pas l'état terminal* **faire**
4   Choisir A_t depuis S_t en utilisant la politique comportementale spécifiée par Q;
5   Exécuter A_t;
6   Observer R_t et S_{t+1};
7   Mettre Q à jour;
8     $S_t \leftarrow S_{t+1}$;
9 **fin**

IV. MÉTHODOLOGIE

L'architecture de la solution proposée, illustrée sur la figure 3 et détaillée dans la suite, repose sur le modèle des processus de décision Markovien. À ce stade du développement, il n'est cependant pas assuré que cette architecture fonctionne. Dans cette architecture, deux entités interagissent l'une avec l'autre : l'environnement et l'agent.

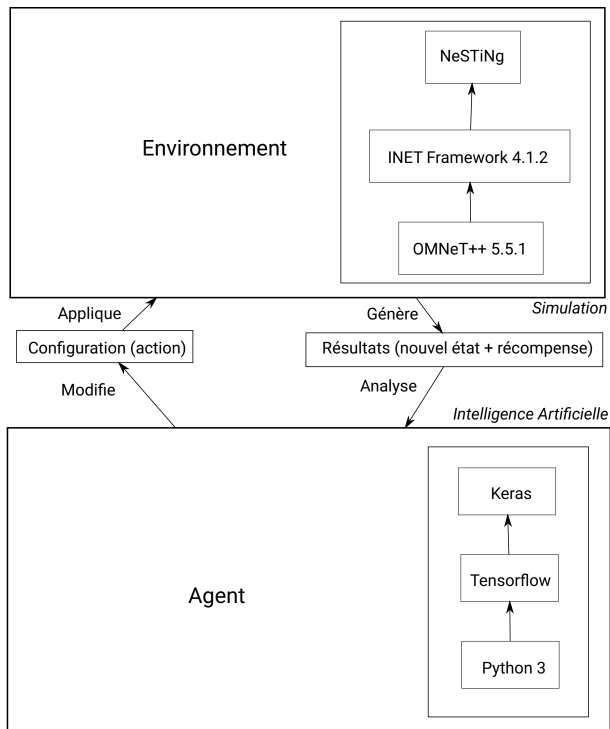

FIGURE 3. L'architecture proposée

A. Environnement

L'environnement est modélisé grâce à une simulation d'un réseau TSN. Pour ce faire, nous utiliserons NeSTiNg [10]. Il s'agit d'une extension apportant les fonctionnalités de TSN au simulateur de réseaux informatique OMNET++/INET.

NeSTiNg est un outil intéressant en cela que la configuration de TSN réside dans des fichiers XML, d'une part, et que les résultats obtenus sont facilement analysables en Python, d'autre part. Ceci permet de gérer l'environnement depuis un script Python qui modifie la configuration, lance la simulation et analyse les résultats. De plus, OMNET++ et NeSTiNg sont des outils reconnus lorsqu'il s'agit d'effectuer des simulations de réseaux TSN.

La simulation permet d'entrainer et d'évaluer l'agent, ainsi que la pertinence de la solution. À terme, si ces résultats sont satisfaisants, l'agent sera bien évidemment confronté à un environnement physique (commutateurs TSN, stations) et réaliste (flux TSN) comparable à un environnement industriel.

Au tout début de la simulation, dans l'état initial, la valeur par défaut de la durée du cycle est fixée arbitrairement à 10ms.

B. Agent

L'agent consiste en un script Python. Nous utilisons plus particulièrement les librairies Tensorflow (pour l'apprentissage) et Keras (pour les réseaux de neurones). À ce stade de la thèse, le but est de savoir si l'architecture proposée plus haut est réalisable. Les technologies choisies (Python, Tensorflow) sont donc les technologies les plus courantes, pas forcément les plus pertinentes. L'optimisation viendra dans un second temps. On appelle *hyperparamètres* les paramètres servant à contrôler l'apprentissage. Ces hyperparamètres sont, d'une part, l'architecture du réseau de neurones (nombre de couches, de neurones) et d'autre part les fonctions d'activation et de perte. La plupart des choix faits relativement aux hyperparamètres du réseau de neurones proviennent de l'exemple de code fourni par Keras.

1) Architecture du réseau de neurones: Un réseau de neurones consiste en des nœuds appelés *neurones* répartis dans des *couches*. Chaque neurone est relié à au moins un neurone de la couche précédente et à au moins un neurone de la couche suivante. On distingue trois types de couches : les couches d'entrée, de sortie et cachées. Dans le cas de Deep Q-learning, le nombre de neurones de la couche d'entrée correspond au nombre de variables dans l'état (10), tandis que le nombre de neurones dans la couche de sortie correspond au nombre d'actions possibles (4). Le nombre de couches cachées, ainsi que le nombre de neurones qu'elles contiennent, sont paramétrables. Nous allons commencer (arbitrairement) avec 2 couches, chacune contenant 10 neurones. Ces « hyperparamètres » évolueront au fil des expérimentations et de lectures futures.

2) Fonction d'activation: La fonction d'activation permet de rendre le fonctionnement de chaque neurone non-linéaire. Cela permet au réseau de neurones d'apprendre des schémas plus complexes. Le choix de la fonction d'activation est aussi un hyperparamètre, car cela revient à un problème d'optimisation. *ReLU* (pour Rectifier Linear Unit) est la fonction d'activation retenue pour commencer.

3) Fonction de perte: La fonction de perte permet de calculer les erreurs du modèle. Nous utilisons une fonction de perte appelée *fonction de perte de Huber*. Après avoir calculé l'erreur, le résultat est propagé dans le réseau : on appelle cela la *rétropropagation*. L'algorithme que nous utilisons pour cela s'appelle *Adam*. Adam est un algorithme du gradient qui permet d'optimiser le résultat de la fonction de perte, plus précisément, il permet de trouver le minimum de cette fonction.

C. Lien entre agent et environnement

La tâche la plus difficile consiste à coder l'interaction entre l'agent et l'environnement. Dans la plupart des exemples de code disponibles sur Internet, le lien est fait grâce à une librairie appelée OpenAI Gym. Malheureusement, cette facilité n'existe pas pour OMNET++. Il y a donc deux solutions. La première consiste à inclure OpenAI Gym dans OMNET++ en faisant quelque chose de similaire à [11]. La seconde, moins élégante mais plus simple sur le court terme, consiste à coder à la main le lien entre l'agent et l'environnement en se basant sur les fichiers XML de configuration et sur les résultats de la simulation. C'est cette seconde solution qui a été retenue dans un premier temps.

Plus précisément, le fonctionnement est le suivant. L'agent modifie la configuration de Qbv dans le fichier XML, puis lance la simulation. Quand elle est terminée, l'agent analyse les résultats de la simulation, en particulier la latence et la gigue. Il détermine sa récompense en fonction de ces résultats et met à jour l'état, puis le cycle recommence.

V. Conclusion

Dans cet article, nous avons proposé une architecture permettant de configurer dynamiquement des réseaux TSN grâce aux techniques de l'IA.

La première étape des travaux à venir est de faire une implémentation et de faire varier les hyperparamètres du réseau de neurones. Ensuite, il s'agira de complexifier le modèle. Ces travaux demanderont d'essayer de nouveaux algorithmes d'apprentissage, d'ajouter d'autres normes TSN, de rendre le réseau dynamique et, à la fin, de rendre le réseau réaliste.

Références

[1] J. Prados-Garzon, T. Taleb, and M. Bagaa, "LEARNET : Reinforcement learning based flow scheduling for asynchronous deterministic networks," in *ICC 2020-2020 IEEE International Conference on Communications (ICC)*. IEEE, 2020, pp. 1–6.

[2] M. Gutiérrez, A. Ademaj, W. Steiner, R. Dobrin, and S. Punnekkat, "Self-configuration of IEEE 802.1 TSN networks," in *2017 22nd IEEE International Conference on Emerging Technologies and Factory Automation (ETFA)*. IEEE, 2017, pp. 1–8.

[3] N. Navet, T. L. Mai, and J. Migge, "Using machine learning to speed up the design space exploration of ethernet TSN networks," University of Luxembourg, Tech. Rep., 2019.

[4] J. Hofmann, "Deep reinforcement learning for configuration of time-sensitive-networking," Bachelor's thesis, Universität Würzburg, 2020.

[5] J. Farkas, L. L. Bello, and C. Gunther, "Time-sensitive networking standards," *IEEE Communications Standards Magazine*, vol. 2, no. 2, pp. 20–21, 2018.

[6] IEEE 802.1 Working Group, "IEEE Standard for Local and metropolitan area networks – Bridges and Bridged Networks - Amendment 25 : Enhancements for Scheduled Traffic," *IEEE Std 802.1Qbv-2015 (Amendment to IEEE Std 802.1Q-2014 as amended by IEEE Std 802.1Qca-2015, IEEE Std 802.1Qcd-2015, and IEEE Std 802.1Q-2014/Cor 1-2015)*, pp. 1–57, 2016.

[7] R. S. Sutton and A. G. Barto, *Reinforcement learning : An introduction*. MIT press, 2018.

[8] V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra, and M. Riedmiller, "Playing atari with deep reinforcement learning," *arXiv preprint arXiv :1312.5602*, 2013.

[9] V. Mnih, K. Kavukcuoglu, D. Silver, A. A. Rusu, J. Veness, M. G. Bellemare, A. Graves, M. Riedmiller, A. K. Fidjeland, G. Ostrovski *et al.*, "Human-level control through deep reinforcement learning," *nature*, vol. 518, no. 7540, pp. 529–533, 2015.

[10] J. Falk, D. Hellmanns, B. Carabelli, N. Nayak, F. Dürr, S. Kehrer, and K. Rothermel, "NeSTiNg : Simulating IEEE time-sensitive networking (TSN) in OMNeT++," in *2019 International Conference on Networked Systems (NetSys)*. Garching b. München, Germany : IEEE, 2019, pp. 1–8.

[11] P. Gawłowicz and A. Zubow, "Ns-3 meets openai gym : The playground for machine learning in networking research," in *Proceedings of the 22nd International ACM Conference on Modeling, Analysis and Simulation of Wireless and Mobile Systems*, 2019, pp. 113–120.