Session des doctorants : Actes

Comité de programme :

- Matheus Ladeira, LIAS/ISAE-ENSMA (Responsable)
- Ill-Ham Atchadam, Lab-STICC/ Université de Bretagne Occidentale
- Quentin Bailleul, IRT St-Exupery
- Thomas Beck, Airbus Defence and Space
- Pierre Bouvier, Convecs/INRIA Grenoble
- Soulimane Kamni, LIAS/ISAE-ENSMA

Table des matières

1 Automatic predictable C code generation of machine learning models for avionics systems 5

2 Deep Q-learning pour la gestion de réseaux déterministes 9

3 Vérification d'applications temps-réel basées sur le paradigme de Logical Execution Time (LET) 13

4 Effets des Régulateurs de Trafic dans les Réseaux en Temps Contraint 17

5 Une approche de génération automatique de configuration basée sur les modèles pour les réseaux TSN 21

6 Towards robust network synchronization with IEEE 802.1AS 25

7 Mutual perturbations of Fprime applications in a space context 29

8 Allocation dynamique de cache mémoire dans un processeur multi-coeur avec des tâches à criticité multiple 35

9 Dimensionnement des buffers en nombre de trames dans l'Ethernet commuté 39

Automatic predictable C code generation of machine learning models for avionics systems

Iryna De Albuquerque Silva*†, Thomas Carle‡, Adrien Gauffriau§, Claire Pagetti†
*ANITI, †ONERA, ‡IRIT, §Airbus

***Abstract*—Machine learning applications have been gaining considerable attention in the field of safety-critical systems. However, a lot of research and engineering development has yet to be conducted in order to respond to the particular constraints of the avionics industry and widely embed these algorithms in aircraft. The scope of this work is the safe real-time implementation of neural networks on embedded avionic platforms. Essentially, we need to guarantee three particular properties in the embedded function:**

1) ***semantic preservation (or reproducibility)*, that is the capacity to replicate the functional behavior observed during design phase;**
2) ***predictability*, that is the capacity to assess its WCET (Worst Case Execution Time);**
3) ***efficiency*, that is the capacity to correctly use hardware resources and be executed quickly.**

We have defined a generic generation of C code from TensorFlow/Keras trained fully-connected neural networks that preserves the semantics and which can be assessed with OTAWA. We have compared our results with those of the Keras2C tool and showed that we are equally efficient with a simpler and time-analyzable code.

I. Introduction

The use of artificial intelligence approaches is already of vital importance in many research areas. In particular, when embedded in aircraft systems, intelligent algorithms could help in tasks such as navigation, predictive maintenance and air traffic control, improving safety and saving environmental resources. Nonetheless, not much progress has been made in embedding machine learning solutions in safety-critical systems as most of those applications do not reach classical safety confidence levels and are not implemented with accepted development process [1], [2].

This work is restrained to off-line learned feed-forward neural networks (referred to simply as neural networks or NN subsequently). The design of such neural networks is done by defining the structure (that is the number of layers, neurons in each layer, activation functions), choosing the training data set and using a learning framework such as TensorFlow [3]. The result of the design is called the *inference model*: it comprises a neural network with its hyper-parameters (e.g. weights, biases, activation functions) and nothing else (e.g. no element from training). The implementation consists in coding the inference model in an adequate programming language and porting the code on the hardware target.

A. Challenges for embedding ML code

The first challenge brought by the implementation is the *semantic preservation*: the reproducibility of the behavior observed at the end of the design when executing the *inference model* within the training tool and on the hardware target. In TensorFlow/Keras, this is done by calling the *predict* method. Even though the mathematical description of neural networks is given in the literature, the training tools do not encode the operations in the same manner. This is particularly true for convolutions, where some implementations start from the top left and some from the bottom right of the matrix, or compute the padding in a different way. This has been observed in [4] and could be reproduced by experimenting with the frameworks. There are lots of works tackling the interoperability among frameworks, by proposing conversion tools [4] or defining open source formats such as ONNX [5] but, so far, the description is still incomplete. Tool vendors such as eIQ [6] from NXP or KaNN [7] from Kalray face this problem, and cannot support all input formats because this requires to reverse engineer the libraries present in training frameworks. In this work, we focus on TensorFlow/Keras framework. One of our objectives is to preserve the semantics of the generated code, meaning that we want the exact same behavior and, if not possible, the closest one (this will be refined later in the paper).

The second challenge is the *predictability*: capacity to assess the WCET (Worst Case Execution Time) [8] of a sequential code. In the ML literature, most of the implementations are done on GPUs or TPUs with an engine such as TensorFlow that interprets the *computation graph*, i.e., a graph describing the mathematical structure of the neural network, and such an interpreter uses dynamic memory allocation. As we focus on safety-critical domain and more specifically avionics, such an approach is not practical for two reasons. First, the hardware targets that are compatible with certification are not those mentioned earlier. We thus focus on general purpose multi-core commercial off-the-shelf (COTS) hardware such as the T1042 from NXP or the Coolidge from Kalray [9]. Second, the programs, including the application, the RTOS and the runtime, must be *predictable*. Therefore, embedding the ML engine is not practical. There are some initiatives to make such runtime predictable such as eIQ and KaNN. However, there is still a large amount of work and proof to show the capability to compute a WCET for these tools. This is the reason why we target a more classical static approach which consists in generating an equivalent C code to execute the model (no interpretation) such as proposed in [10].

Lastly, we know that producing C code on legacy avionic hardware will make the execution *inefficient* compared to GPU execution. However, our purpose is to efficiently use the resources in order to be predictable and efficient. This last challenge will not be discussed in the paper.

B. Contributions

We managed to generate compatible and generic C code for fully-connected feed-forward neural networks trained in Keras. We validated the generated code by comparing the predictions in C with those provided by the learning framework. We observed an almost equivalent behavior. To assess the predictability we computed the WCET with OTAWA [11]. To do so it is necessary to choose a hardware target and we picked one among the micro-architectures models available in OTAWA. We chose an ARM-based single-core target executed in bare-metal. Finally, we made some comparisons with Keras2C code and proved our approach beneficial.

The rest of this paper is organized as follows: in Section II we discuss the state of the art in this domain. Section III presents our contribution to the challenge of embedding neural networks in aircraft. Finally, in Section IV we evaluate our experimental results and Section V concludes the paper.

II. Related work

There are mainly three methods that could fit our problem statement, that is the predictable code generation for avionic-compatible multi-core COTS hardware for machine learning.

The first work [10] is guided by avionics constraints as well and, in order to provide an efficient implementation of Deep Neural Network (DNN) inference models, the authors developed an automatic code generator that allows preserving semantics of the trained machine learning model. However, the code generation tool is not extensively described nor made available.

The second is Keras2C [12]. This method consists in a library to convert Keras/TensorFlow neural network models into real-time compatible C code, supporting a wide range of Keras layers and relying only on C standard library functions. In the evaluation section we will compare our results with Keras2C with respect to semantic preservation, predictability and efficiency.

The study of [13] also investigates a predictable implementation of neural networks for safety-critical cyber-physical systems. They embed the Keras2C code on Patmos, a time-predictable processor, which is part of the larger T-CREST [14] project. The software tool-chain of the latter includes a LLVM-based compiler and the Platin tool for WCET analysis.

The third approach is the use of a generic compiler framework that tries to be agnostic of the inputs. Among the available tools, we can mention TVM [15]. TVM is a tool capable of compiling machine learning models from different popular frameworks and generating specific low-level optimized code for a diverse set of hardware back-ends. We can also mention N2D2 [16], where the authors explore how approximation techniques can improve the performance and energy efficiency of hardware accelerators in machine learning applications. We have not yet assessed these tools and this study will be done as future work.

III. Automatic Code Generation

Although our main goal is to exploit various neural network architectures, we started with fully- connected neural networks as they are the simplest and are often a sub-part of more complex ones (e.g. Convolutional Neural Networks).

Definition 1 (Feed-Forward Neural Network): A feed-forward neural network can be formalized as a tuple $S =< X, Y, L, N, \sigma, W, B >$, where:

- *$X \in \mathbb{R}^m$ is a finite collection of m inputs of the neural network;*
- *$Y \in \mathbb{R}^p$ is a finite collection of p outputs of the neural network;*
- *L denotes the numbers of layers in the neural network, $L > 2$, l_0 represents the input layer, $l_{(L-1)}$ is the output layer and those in between are called hidden layers;*
- *N is a set of neurons, where k_l represents the number of neurons in layer l;*
- *σ is the activation function implemented by the neurons;*
- *W is set of weights connecting the layers, wherein $W_l \in \mathbb{R}^{k_l \times k_{l-1}}$ are the weights connecting layer $l-1$ to layer l.*
- *B is the set of biases, wherein $B_l \in \mathbb{R}^{k_l}$ are the biases of layer l.*

An example of a feed-forward neural network can be observed in figure 1.

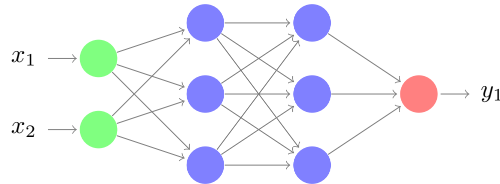


Fig. 1. Example of a feed-forward neural network, composed of 2 hidden layers with 2 neurons in each, 2 inputs and 1 output.

We used the Keras framework for model training, which could export the representation of the computation graph in several formats (e.g. *.h5*, which is a proprietary Tensorflow binary format, or *.pb* protobuf [17] and *.ONNX*, that are both language-independent and open source). Instead, we used *.json* format to store our trained model, mainly because it is an agnostic format, fully textual and we can master the semantics of the model by including any additional information.

The chosen approach consists of:

- hard-coding several C files which remain unchanged for any feed-forward neural network. More precisely, we have defined an *inference* function that computes layer by layer the propagation of any input. We have also encoded the activation functions (*ReLU, sigmoid or tanh*) with the standard C mathematical library.
- generating for each feed-forward neural network some header files including the layer sizes and the number of neurons per layer, and another C file describing all the global variables such as the weights.

Our purpose was to be generic and compress the code as much as possible by minimizing the loops for instance. As a result, there are 93 lines in the hard-coded files.

The generation of the header and global variables C files, that are model dependent, is done automatically: a python script reads the model architecture and parameters from the $.json$ file and writes them as constants in the C files. Together with the library of activation functions we have the low-level translation of the inference algorithm, as depicted in Figure 2.

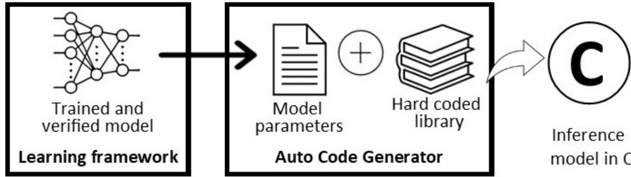


Fig. 2. Building blocks of the auto code generation process.

IV. Evaluation

To experiment and evaluate the C code generation, we have defined a large test campaign by varying several parameters:

- number of layers: ranging from 3 to 100;
- number of inputs: ranging from 2 to 256;
- number of outputs: ranging from 1 to 256;
- number of neurons in each hidden layer: ranging from 2 to 512;
- activation functions: among *ReLU, sigmoid, hyperbolic tangent (tanh)* and *linear*;
- data type: floating-point double precision (FP64), floating-point single precision (FP32) and integer (INT);

Overall, we experimented 100 different model architectures and for every model the testing dataset was composed of 1,000 samples. Besides creating our own experiments, we also chose to evaluate our automatic code generation on the neural network implementation of the airborne collision avoidance system for unmanned aircraft (ACAS-Xu) [18], [19]. ACAS-Xu system maps input variables - information from sensors measurements - to 5 action advisories - represented by scores. We thus considered 10 feed-forward neural networks (divided in 2 types of architecture) of the ML implementation of a reduced footprint ACAS-Xu system provided by [20].

Some architectures will be underlined subsequently in the paper:

- **architecture 1**: 16 hidden layers, 512 neurons in every hidden layer, 256 inputs, 256 outputs, *ReLU* as activation function and FP64;
- **architecture 2**: 8 hidden layers, 64 neurons in every hidden layer, 8 inputs, 8 outputs, *tanh* as activation function and FP64; ;
- **architecture 3**: 8 hidden layers, (16, 16, 32, 64, 128, 64, 32, 16, 4, 1) neurons, *sigmoid* as activation function and FP32;
- **architecture 4**: 5 hidden layers, (5, 128, 128, 64, 32, 16, 5) neurons, *ReLU* as activation function and INT;
- **architecture 5** (ACAS-Xu *regular50*): 6 hidden layers, (5, 50, 50, 50, 50, 50, 50, 5) neurons, *ReLU* for all the hidden layers, *linear* activation for output layer and FP32;
- **architecture 6** (ACAS-Xu *decreasing128*): 5 hidden layers, (5, 128, 128, 64, 32, 16, 5) neurons, *ReLU* for all the hidden layers, *linear* activation for output layer and FP32;

TABLE I. NN architectures with the results for semantic preservation and predictability.

Benchmark	Maximum Error	WCET [cycles]
Architecture 1	0	109073386745
Architecture 2	2.220446049250313e-16	111112160
Architecture 3	5.9604645e-08	269646545
Architecture 4	0	90748125
Architecture 5	0	29349530
Architecture 6	4.641999e-03	136677250

Table I summarizes the results obtained for *semantic preservation* and *predictability*. Because the 5 neural networks of each configuration of ACAS-Xu produce very similar results we only present here one per architecture, namely *COC*.

A. Semantic preservation

To validate the correctness of the code generation, we need to prove the *semantic preservation* in first place. To do so, we could have used formal methods (such as Coq [21] as was done for Velus [22]) but instead, we chose to review the code and run a large campaign of tests. We know that it may be less sound but this is an acknowledged way in the certification DO178C [23].

The semantic preservation is then assessed by comparing the predictions of our inference algorithm implemented in C with those provided by Keras. Let x be a vector representing Keras outputs for a given set of inputs and $\tilde{x}$ be the vector of outputs of our C implementation. We define, then, the absolute error in the vector of our prediction as being the L-infinity norm of the vector of differences, i.e., $\|\tilde{x} - x\|_\infty = max_i|(\tilde{x} - x)|_i$. We particularly chose this norm because we are interested in asserting a maximum bound on the error observed for a given testing sample.

From Table I we can see that the maximum error observed varies significantly depending on the structure of the NN, the activation function and the data type used. Particularly, when we have networks with a regular structure (constant number of neurons in hidden layers) using only *ReLU* and FP64, there is no difference between C and Keras predictions. We are a bit surprised that still using only *ReLU* the error is different from zero when the NN structure is no longer regular. We will investigate further where this discrepancy comes from. However, as expected, independently of the structure of the NN, when using *ReLU* on integers there are no errors. When the activations functions are *tanh* or *sigmoid*, there are some slight differences, up to 10^{-16} when in double-precision and 10^{-8} in single-precision. This can be explained by the fact that python and C do not use the same math library.

B. WCET analysis

We compiled the generated C code for bare-metal ARM targets and used OTAWA to compute the WCET of the resulting binaries. In parallel to the binary code we provided to OTAWA a file with *flow facts* information. Essentially, the

TABLE II. COMPARISON TO KERAS2C WITH RESPECT TO MAXIMUM ERROR IN PREDICTION AND AVERAGE EXECUTION TIME.

Benchmark	Our approach: Maximum error	Our approach: Avg. execution time [s]	Keras2C: Maximum error	Keras2C: Avg. execution time [s]
Acas-Xu *decreasing128*	4.641999e-03	8.048300e-04	4.638672e-03	3.221500e-05
Acas-Xu *regular50*	0	3.657400e-04	0	1.982000e-05

flow facts inform about the set of possible execution paths of a program, e.g. loop iteration counts.

The WCET analysis was done over the 1,000 samples of the testing dataset and we can note from Table I that the WCET also depends directly on the number of layers of the neural network, the number of neurons in every layer, the activation function used and the data type of the model. Running OTAWA on our inference function was instantaneous.

C. Comparison to Keras2C

In this section, we compare our approach with Keras2C. A limitation we directly observed in Keras2C is the need to use *dynamic memory allocation* when working with large NNs, which as explained, implies additional certification challenges in terms of verification and is not at all suited for WCET analysis. Additionally, the code generated by Keras2C uses exclusively simple-precision floating-point data (FP32).

We thus focused only on the architectures 4 and 5 which manipulate FP32 only. Table II sums up the performance metrics analyzed.

1) Semantics preservation: When compared to Keras2C we obtain very similar results in terms of semantics preservation, with errors in order of 10^{-3}.

2) WCET analysis: Furthermore, we were not able to assess the WCET with OTAWA for the code generated by Keras2C because it uses functions of libraries for which we do not have the information of flow facts.

3) Efficiency: To measure this metric, we used the *clock()* function of C standard library. It is not at all reliable but it gives, however, some tendencies. As can be seen from Table II, the inference algorithm of Keras2C has an *average execution time* slightly better than ours. Nevertheless, their approach produces binaries notably bigger than those provided by us for the same model. For *ACAS-Xu decreasing128* for example, Keras2C binary is 679kB while our is 160kB.

V. CONCLUSION

Machine learning applications are proven to be useful and are largely used in many domains, however, most of them are not built with avionics constraints in mind. In this work, we presented our approach to automatically reproduce the inference model of a fully-connected neural network in C code, respecting semantic preservation and predictability requirements. In future work we aim to examine the efficiency of our function when actually embedded in target hardware.

REFERENCES

[1] E. Alves, D. Bhatt *et al.*, "Considerations in assuring safety of increasingly autonomous systems," *NASA*, 2018.

[2] S. Bhattacharyya, D. Cofer *et al.*, "Certification considerations for adaptive systems," in *2015 International Conference on Unmanned Aircraft Systems (ICUAS)*, 2015, pp. 270–279.

[3] M. Abadi, A. Agarwal *et al.*, "TensorFlow: Large-scale machine learning on heterogeneous systems," 2015, software available from tensorflow.org. [Online]. Available: https://www.tensorflow.org/

[4] Y. Liu, C. Chen *et al.*, "Enhancing the interoperability between deep learning frameworks by model conversion," *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2020.

[5] J. Bai, F. Lu *et al.*, "Onnx: Open neural network exchange," https://onnx.ai/, 2019.

[6] "Eiq™ ml software development environment," 2020. [Online]. Available: https://www.nxp.com/design/software/development-software/eiq-ml-development-environment:EIQ

[7] Kalray, "Kann platform for high-performance machine learning inference on kalray's mppa® intelligent processor," 2018.

[8] R. Wilhelm, J. Engblom *et al.*, "The worst-case execution-time problem—overview of methods and survey of tools," *ACM Trans. Embed. Comput. Syst.*, May 2008.

[9] Kalray, "Mppa® coolidge™ processor - white paper," https://www.kalrayinc.com/documentation/, 2021.

[10] S. Chichin, D. Portes *et al.*, "Capability to embed deep neural networks: Study on cpu processor in avionics context," in *10th ERTS*, 2020.

[11] C. Ballabriga, H. Cassé *et al.*, "OTAWA: an open toolbox for adaptive WCET analysis," in *8th IFIP WG 10.2 International Workshop, SEUS*, 2010, pp. 35–46.

[12] R. Conlin, K. Erickson *et al.*, "Keras2c: A library for converting keras neural networks to real-time compatible c," *Engineering Applications of Artificial Intelligence*, 2021.

[13] H. Pearce, X. Yang *et al.*, "Designing neural networks for real-time systems," *IEEE Embedded Systems Letters*, p. 1–1, 2020.

[14] M. Schoeberl, S. Abbaspour *et al.*, "T-crest: Time-predictable multi-core architecture for embedded systems," *Journal of Systems Architecture*, vol. 61, no. 9, pp. 449–471, 2015.

[15] T. Chen, T. Moreau *et al.*, "Tvm: An automated end-to-end optimizing compiler for deep learning," 2018.

[16] O. Sentieys, S. Filip *et al.*, "Adequatedl: Approximating deep learning accelerators," in *24th International Symposium on Design and Diagnostics of Electronic Circuits Systems (DDECS 21)*, 2021.

[17] Google, "Protocol buffers," https://developers.google.com/protocol-buffers/, 2001.

[18] M. P. Owen, A. Panken *et al.*, "Acas xu: Integrated collision avoidance and detect and avoid capability for uas," in *38th Digital Avionics Systems Conference (DASC)*, 2019, pp. 1–10.

[19] G. Katz, C. W. Barrett *et al.*, "Reluplex: An efficient SMT solver for verifying deep neural networks," in *29th International Conference in Computer Aided Verification, July 24-28, 2017, Proceedings, Part I*, ser. Lecture Notes in Computer Science, vol. 10426, 2017, pp. 97–117.

[20] M. Damour, F. De Grancey *et al.*, "Towards certification of a reduced footprint acas-xu system: A hybrid ml-based solution," in *40th International Conference on Computer Safety, Reliability, and Security SAFECOMP 2021*, ser. Lecture Notes in Computer Science, 2021.

[21] T. C. D. Team, "The coq proof assistant," https://coq.inria.fr/, 2021.

[22] T. Bourke, L. Brun *et al.*, "A formally verified compiler for lustre," in *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2017, Barcelona, Spain, June 18-23, 2017*, A. Cohen and M. T. Vechev, Eds. ACM, 2017, pp. 586–601.

[23] *DO-178C, Software Considerations in Airborne Systems and Equipment Certification*, RTCA, Inc. Std., 2011.

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.

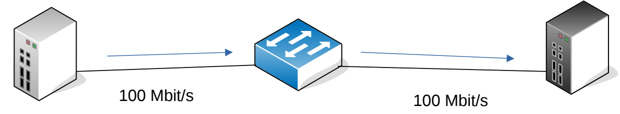


Figure 1. La topologie considérée

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).

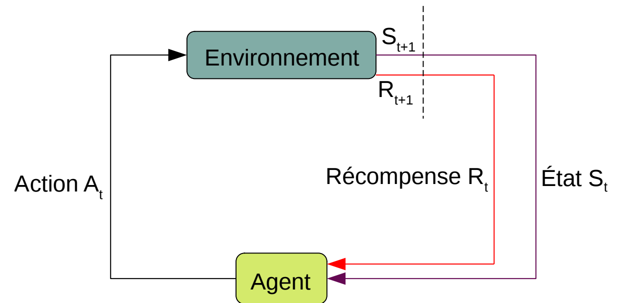


Figure 2. Les interactions entre l'agent et l'environnement dans un MDP

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.

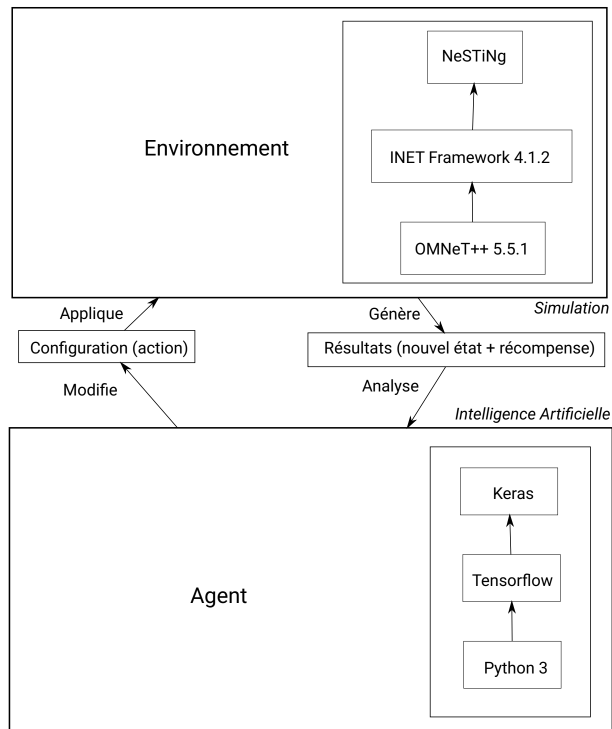


FIGURE 3. L'architecture proposée

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.

Vérification d'applications temps-réel basées sur le paradigme de Logical Execution Time (LET)

Fabien Siron*†, Dumitru Potop-Butucaru†, Robert de Simone†, Damien Chabrol* et Amira Methni*
*Krono-Safe - Massy, France
†INRIA - Paris/Sophia-Antipolis, France
*firstname.lastname@krono-safe.com
†firstname.lastname@inria.fr

***Résumé*—Le design de logiciel de contrôle/commande embarqué dépend de contraintes temporelles strictes. Pour cela, des formalismes et des théories basés sur la notion de temps logique permettent d'abstraire les durées temps-réel qui ne sont pas toujours connues au niveau conception. Cet article propose une comparaison des formalismes synchrones et du paradigme *Logical Execution Time (LET)* dans le but d'adapter au langage industriel *PsyC*, qui est proche du *LET*, des méthodes de vérification issues des langages synchrones.**

I. INTRODUCTION

Le design de logiciel de contrôle/commande embarqué, intéragissant massivement avec leur environnement, dépend de contraintes temporelles strictes. La correction temporelle du système doit être assurée au plus tôt dans le cycle logiciel, en particulier dans le contexte de systèmes critiques. Cependant, les durées temps-réel ne sont pas toujours connues au moment de la conception.

Plusieurs formalismes et théories basés sur la notion de temps logique ont été introduits afin de palier à ce problème. Le design est ainsi divisé en deux phases : d'une part la spécification, la programmation et la vérification basés sur le temps logique imposant des hypothèses sur les temps d'exécution et d'autre part, la validation de ces hypothèses vis à vis de temps physique lié à l'implémentation. Cet article propose une comparaison des formalismes synchrones et du paradigme *Logical Execution Time (LET)* (tous les deux issus de la notion de temps logique), dans le but d'adapter au langage industriel *PsyC*, qui est proche du *LET*, des méthodes de vérification issues des langages synchrones. En effet, bien qu'il y ait eu beaucoup de travaux sur la vérification formelle de langages synchrones [1], à notre connaissance, il n'en existe pas pour les langages complexes basés sur le *LET* tel que *PsyC*.

Dans la première section, nous détaillerons la notion de langage synchrone ainsi que ses différentes variantes. Nous détaillerons ensuite dans la deuxième section le paradigme *LET* ainsi que ses différentes implémentations et nous le comparerons aux approches synchrones. Nous donnerons ensuite une description informelle du langage *PsyC*, qui est une implémentation du paradigme *LET* par la société *Krono-Safe*. Nous terminerons par montrer comment le paradigme synchrone peut être utilisé pour la vérification d'applications basées sur le paradigme *LET*, tel que le *PsyC*.

II. LANGAGES SYNCHRONES

Les langages synchrones permettent d'atteindre les propriétés de déterminisme et de concurrence au travers de calculs élémentaires qui réagissent simultanément et instantanément à chaque tick d'une horloge globale commune. Dans ces langages, le temps n'est pas exprimé explicitement, mais est vu comme une séquence ordonnée de réactions atomiques (ou *instants*) du système ; la notion de temps est donc remplacée par la notion d'ordre, on parle alors de temps logique. L'hypothèse synchrone garantie alors, que si tous les calculs sont effectués avant la prochaine réaction, alors le temps physique peut être ignoré de manière sûre [2].

On peut classer les langages synchrones en deux grandes familles :

- — les langages synchrones *flot de données* (e.g. tel que *Lustre* [3]) qui sont généralement déclaratifs, formés d'équations temporelles sur des flots auquels sont associés des horloges ;
- — et les langages synchrones *flot de contrôle* (e.g. tel qu'*Esterel* [4]) qui sont généralement impératifs, permettant la concurrence impérative (i.e. l'existence de plusieurs pointeurs d'instructions différents coexistants au même instant) ainsi que la préemption qui permet d'interrompre une exécution.

Il convient cependant de nuancer les avantages des langages synchrones. Leur compilation est assez complexe et ne permet pas, généralement, de générer du code parallèle. De plus, les temps d'exécution physiques sont généralement limités au temps de réaction du système, appelé pire temps de réaction (WCRT). Cela complique la modélisation de système multi-périodiques, qui pourtant, aujourd'hui, deviennent de plus en plus nombreux.

Des approches plus récentes permettent toutefois de palier à ces problèmes en affaiblissant le paradigme synchrone tout en maintenant sa propriété de déterminisme. On peut citer l'algèbre synchrone *CoReA* de Boniol (*Communicating Reactive Automata* [5], inspiré de *SCCS* [6]), qui développe un modèle synchrone dit *faible* (en opposition au modèle synchrone classique, dit *fort*). Dans ce modèle, les calculs réagissent aux ticks d'une horloge globale, comme dans le modèle *synchrone fort*, mais produisent le résultat de leur réaction au prochain instant. Plus récemment, les travaux de Forget [7] introduisent un langage synchrone déclaratif multi-périodique, appelé *Prélude*. Ce langage, basé sur *Lustre*, développe l'idée d'une hypothèse synchrone *relachée*, c'est à dire que chaque flot, lors de sa réaction, doit finir son calcul

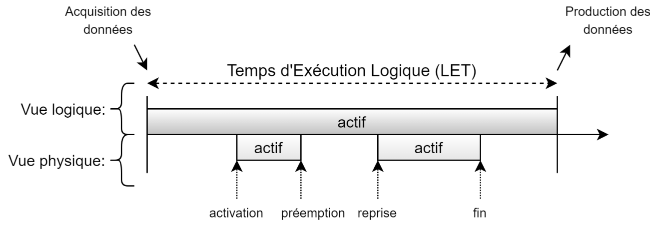


FIGURE 1. Modélisation d'un intervalle *LET*

avant sa prochaine activation. Cela implique que chaque flot à sa propre notion d'instant et permet ainsi la modélisation de système multi-périodiques. Affaiblir le modèle synchrone permet aussi d'éviter les problèmes de causalité, inhérents à ces approches [8].

III. LOGICAL EXECUTION TIME

Le paradigme *Logical Execution Time (LET)* [9] abstrait, de manière similaire au paradigme synchrone, le temps physique, via une succession d'instants logiques. Cependant, le calcul n'est pas considéré instantané mais durant un intervalle de temps logique (ou intervalle *LET*) défini par un instant de départ et de fin. Les communications sont ainsi effectuées sur ces deux instants : les entrées sont consommées sur l'instant de départ et les sorties sont produites sur l'instant de fin (voir figure 1). En d'autre terme, la durée effective d'une réaction est abstraite par le délai entre entrées et sorties. Cela se justifie du fait que le temps observable d'une fonction n'est pas réellement son temps effectif de calcul, mais le temps entre la prise en compte des entrées et la production des sorties.

En considérant que le *LET* étend le concept de temps logique avec la notion de durée, cela permet une plus grande variation des temps d'exécution. Cela permet ainsi de traiter la problèmatique des systèmes multi-périodiques et de faciliter la compilation vers du code parallèle (i.e. multi-tâche, multi-coeur . . .). De plus, les propriétés de déterminisme et de concurrence issues du paradigme synchrone sont préservées du fait que toutes les communications sont effectuées sur des dates prédéfinies.

Bien que le *LET* soit généralement classé dans les modèles de calcul temporisés [10] du fait de sa proximité avec l'architecture timed-triggered (TTA) concue par Kopetz [11], les instants délimitants les intervalles sont relatifs à une l'horloge commune globale qui est logique. Ainsi, cette horloge peut aussi rythmer les réactions du système de manière évènementiel. Le paradigme *LET* est par conséquent basé sur un temps logique, tel que le paradigme synchrone, et n'est donc pas explicitement temporisé. *Giotto* [12] est une implémentation du *LET* avec des tâches simplement périodiques basées sur le temps physique. Il est étendu au travers de bases de temps évènementielles dans le langage *xGiotto* [13] supportant les tâches non périodiques. Ce langage propose une généralisation du paradigme *LET* acceptant des intervalles *LET* de taille variable. Le langage *PsyC* [14] supporte lui aussi des tâches non périodiques basées sur une base de temps logique. Il est développé dans la prochaine section.

On peut remarquer que le paradigme *LET* partage de

```
clock 2MS = 2 * MS;

temporal int tv_signal = 0 on MS;
temporal int tv_threshold = 0 on 2MS;

agent Threshold(uses realtime, starttime 2 MS)
{
  consult tv_signal $ 2;
  display tv_threshold;
  body start
  {
    tv_threshold = $[0]tv_signal > THRESHOLD ||
                   $[1]tv_signal > THRESHOLD;
    advance 1 with MS;
    /* boucle infinie sur le body start */
  }
}
```

Code 1. Exemple d'un agent *PsyC*

grandes similarités avec le paradigme synchrone. Tous deux permettent de préciser des instants en fonction d'une horloge globale commune aux différentes tâches. Dans le *LET*, contrairement au synchrone, les exécutions ne sont toutefois pas instantanées mais bornées par l'instant de fin de l'intervalle. Il convient cependant de noter qu'une exécution instantanée des calculs est une abstraction correcte du *LET* du fait que le déterminisme est assuré par le modèle de communication et non la durée des réactions. Le modèle de communication peut être abstrait par un délai systématique des communications produites par une tâche sur le prochain instant d'activation de celle-ci. Le paradigme *LET* peut ainsi être abstrait par le paradigme synchrone avec un délai systématique des communications, en d'autre termes, le paradigme synchrone dit *faible* tel que décrit par *CoReA*. Cette abstraction permet, par conséquent, de simplifier le modèle et de faciliter la vérification.

IV. PSYC

Le langage *PsyC* est une implémentation d'une forme généralisée du paradigme *LET* (similairement à *xGiotto*) dédiée aux applications temps-réel critiques. Il a été conçu comme un sur-ensemble du langage C afin de faciliter l'intégration d'applications conçues avec ce langage. Le langage est implémenté dans la suite *ASTERIOS* qui est produite par la société *Krono-Safe*. Cette suite outillée permet le développement d'applications aéronautiques critiques certifiées au plus haut niveau de criticité (DAL-A, DO-178C).

Une application *PsyC* est composée de tâches appelées *agents* qui sont des unités de calcul séquentielles formées d'une suite d'intervalles *LET*. Le contenu d'un agent est composé de code C où une instruction spéciale, `advance n c` dénote les bornes des intervalles *LET* en avançant la date logique de n ticks d'une horloge c dérivée d'une source temporelle unique. Le déterminisme est issu du principe de visibilité des communications inter-tâches. Les données sont timestampées à la date où elles sont émises. Ainsi, au sein d'un intervalle *LET*, les données externes visibles sont celles dont le timestamp est inférieur ou égal à la date logique courante, tandis que les données produites deviennent visibles

au prochain instant d'activation défini par la prochaine instruction `advance` (i.e. l'instant clôturant l'intervalle courant). Au niveau de l'application, les *agents* sont composés en parallèle de manière synchrone et communiquent entre eux principalement via un mode de communication échantillonné appelé *variable temporelle* (VT). L'écriture d'une VT se fait comme une affectation classique mais la lecture se fait via l'expression `$[n]tv` qui dénote le n$^{\text{ième}}$ dernier élément de la VT échantillonnée sur son horloge.

Le code 1 décrit un *agent PsyC* possédant une variable temporelle en entrée échantillonné sur la milliseconde (horloge MS) et une autre variable temporelle en sortie échantillonné sur une horloge 2 fois plus lente (horloge $2MS$) dénotant le dépassement d'un seuil. La déclaration d'horloge `clock` permet de spécifier l'horloge $2MS$ en fonction de l'horloge MS. Les deux variables temporelles sont ensuite déclarées avec une valeur d'initialisation ainsi que leurs horloges respectives. L'agent démarre à la date 2 ms et actualise la variable temporelle de sortie toutes les 1 ms en spécifiant si le seuil a été dépassé.

V. Vérification

A. Observateurs

Nous nous intéressons ici aux propriétés de sûreté des systèmes temps-réel telles que des latences ou bien des motifs de cadencement. Classiquement, ces propriétés peuvent être modelisées via des logiques temporelles telles que *LTL* (logique temporelle linéaire) ou *CTL* (logique temporelle arborescente) [15]. Toutefois, issu des travaux sur les langages synchrones, les observateurs permettent de représenter les propriétés directement dans le formalisme utilisé par le système [1]. Les observateurs détectent les mauvaises exécutions du système (i.e. les exécutions partielles qui invalident une propriété du système). Cela permet, d'une part, d'avoir une homogénéité entre la représentation des propriétés et du système, et d'autre part, de vérifier les propriétés dynamiquement (via du *Runtime Monitoring* [16]) ou bien statiquement (via du *Model-Checking* [1] [15]).

Le code 2 propose un exemple d'observateur vérifiant, sur l'*agent* décrit dans le code 1 la propriété suivante : lorsque le signal dépasse un seuil, alors le signal de sortie est actif lors de l'instant suivant ou celui d'après. Cela se traduit en logique temporelle par la formule suivante [15] :

$$\Box(P \implies \bigcirc(Q \vee \bigcirc Q)) \tag{1}$$

où $P \overset{\text{def}}{=} tv_signal > THRESHOLD$ et $Q \overset{\text{def}}{=} tv_threshold = true$. Cette forme typique spécifie une latence entre l'évènement P et Q d'au minimum un instant et d'au maximum deux instants. Il faut le lire de la manière suivante : *il est toujours vrai que lorsque P est vrai alors Q est vrai à l'instant suivant ou celui d'après*. Le symbole $\Box$ dénote une propriété vraie sur tous les instants et le symbole $\bigcirc$ dénote une propriété vraie sur l'instant suivant. Il convient toutefois de préciser que cette équivalence ne prend pas en compte le *starttime* (i.e. la date de départ de l'*agent*).

```
agent Observer(uses realtime, starttime 2 MS)
{
  consult tv_signal $ 1;
  consult tv_threshold $ 1;
  body start {
    if ($[0]tv_signal > THRESHOLD) {
      advance 1 MS;
      if ($[0]tv_threshold != 1) {
        advance 1 MS;
        if ($[0]tv_threshold != 1)
          ast_error_raise(/* code d'erreur */);
      }
    } else {
      advance 1 MS;
    }
  }
}
```

Code 2. Exemple d'un observateur reprensenté sous forme d'agent *PsyC*

B. Vérification Dynamique via Runtime Monitoring

Les propriétés peuvent être vérifiées directement à l'exécution en considérant les observateurs comme des tâches standards. Le système doit toutefois être doté d'un mécanisme de gestion d'erreur à l'exécution. Dans le cas du *PsyC*, l'instruction `ast_error_raise` permet de lever une erreur au sein d'un *agent* en y associant une sanction (e.g. extinction de la tâche, extinction du système entier ...).

C. Vérification Statique via Model Checking

Les observateurs peuvent aussi servir à la vérification statique via du *model-checking*. En reprenant l'analogie décrite plus haut entre *LET* et synchrone, on peut proposer une traduction du modèle et de son observateur dans un formalisme synchrone. Le langage *PsyC* étant principalement impératif et flot de contrôle, *Esterel* est tout à fait adapté car il partage ces deux aspects.

Les codes 3 et 4 décrivent les traductions respectives de l'agent et de l'observateur. Tout deux partagent une structure similaire où l'instruction `advance` (ainsi que `starttime`) sont remplacées par des instructions `await`. La différence principale réside cependant au niveau des communications où leur émission est systématiquement retardées sur l'instant correspondant à la fin de l'intervalle *LET*. On considère ici que les variables temporelles (VT) sont traduites en signaux valuées (en *Esterel* la valuation des signaux est persistente). Pour une VT `temporal`, le signal *Esterel* `temporal` dénote le signal d'écriture (non échantillonné) et les signaux `temporal_n` dénotent les signaux échantillonnés d'index n accessibles par les lecteurs. Ainsi, dans le code 3, `tv_signal_0` dénote le dernier échantillon de `tv_signal` (échantillonné sur le dernier tick de l'horloge MS) et `tv_signal_1` dénote l'avant dernier échantillon. Par manque de place, les traductions des variables temporelles ne sont pas données. Il s'agit toutefois simplement de module d'échantillonnage classique.

L'erreur levée par l'observateur est remplacée par un signal dénotant un mauvais état de celui-ci (i.e. une erreur). Lors de la compilation *Esterel* permet de générer un automate modélisant tout le système (observateur compris). Il suffit alors d'une analyse d'accessibilité sur l'état d'erreur de l'observateur. Si

```
module Agent:
  input MS;
  input tv_signal_0 : integer;
  input tv_signal_1 : integer;
  output tv_threshold : boolean;
  await 2 MS;
  loop
    if ?tv_signal_0 > THRESHOLD or
       ?tv_signal_1 > THRESHOLD then
      await 1 MS;
      emit tv_threshold(true)
    else
      await 1 MS;
      emit tv_threshold(false)
    end if;
  end loop
end module
```

Code 3. Traduction de l'*agent PsyC* en *Esterel*

```
module Observer:
  input MS;
  input tv_signal_0 : integer;
  input tv_threshold_0 : boolean;
  await 2 MS;
  loop
    if ?tv_signal_0 > THRESHOLD then
      await 1 MS;
      if ?tv_threshold_0  else
        await 1 MS;
        if ?tv_threshold_0  else
          emit Error
        end if
      end if
    else
      await 1 MS
    end if
  end loop
end module
```

Code 4. Traduction de l'observateur de *PsyC* en *Esterel*

cet état est atteignable, alors il peut y avoir une exécution ne respectant pas la propriété spécifiée par l'observateur, sinon, cette propriété est garantie pour toute exécution.

Bien qu'il suffit d'un parcours sur l'automate généré afin de vérifier la propriété, cette approche souffre quand même du problème de l'explosion combinatoire. L'automate généré grandi exponentiellement vis à vis du nombre d'état de chaque tâche. Toutefois, la composition synchrone permet de limiter drastiquement l'explosion combinatoire. D'autre part, la compilation d'*Esterel* permet aussi d'obtenir l'automate sous forme implicite (i.e. via des systèmes d'équations) et permet ainsi la vérification dites "symbolique" qui ne présente pas ces problèmes.

VI. Conclusion

Le paradigme *LET* permet ainsi d'assurer déterminisme et concurrence tout en permettant une compilation simple et une analyse d'ordonançabilité fine. Toutefois, aujourd'hui il n'existe pas réellement d'approche de vérification formelle des langages implémentant ce paradigme. Les modèles synchrones peuvent être utilisés pour abstraire les langages basés sur le *LET* et les techniques de vérification actuelles peuvent être réutilisées. Dans l'exemple utilisé dans cet article, une analyse d'accessibilité de signaux d'erreurs est effectué sur l'automate (explicite) généré par le compilateur d'*Esterel*. La taille de celui-ci pouvant grossir rapidement, une perspective est l'utilisation de représentations implicites d'automate ce qui permettrait de traiter des applications de tailles supérieures. D'autre part, la comparaison entre synchrone et *LET* sera développée plus précisément dans un travail futur.

Références

[1] P. Raymond, "Synchronous Program Verification with Lustre/Lesar," 2010.

[2] A. Benveniste and G. Berry, "The synchronous approach to reactive and real-time systems," *Proceedings of the IEEE*, vol. 79, pp. 1270 – 1282, 10 1991.

[3] D. Pilaud, N. Halbwachs, and J. Plaice, "Lustre : A declarative language for programming synchronous systems," in *Proceedings of the 14th Annual ACM Symposium on Principles of Programming Languages (14th POPL 1987). ACM, New York, NY*, vol. 178, 1987, p. 188.

[4] F. Boussinot and R. De Simone, "The esterel language," *Proceedings of the IEEE*, vol. 79, no. 9, pp. 1293–1304, 1991.

[5] F. Boniol, "CoReA : A synchronous calculus of parallel communicating reactive automata," in *PARLE'94 Parallel Architectures and Languages Europe*. Springer, 1994.

[6] R. Milner, "Calculi for synchrony and asynchrony," *Theoretical computer science*, vol. 25, no. 3, pp. 267–310, 1983.

[7] J. Forget, F. Boniol, D. Lesens, and C. Pagetti, "A Multi-Periodic Synchronous Data-Flow Language," in *11th IEEE High Assurance Systems Engineering Symposium*, Dec. 2008, p. 251.

[8] F. Boussinot and R. De Simone, "The sl synchronous language," *IEEE Transactions on Software Engineering*, vol. 22, no. 4, pp. 256–266, 1996.

[9] C. Kirsch and A. Sokolova, "The logical execution time paradigm," in *Advances in Real-Time Systems*. Springer Berlin Heidelberg, 10 2012, pp. 103–120.

[10] E. A. Lee and S. A. Seshia, *Introduction to embedded systems : a cyber-physical systems approach*, second edition ed. Cambridge, Massachuetts : MIT Press, 2017.

[11] H. Kopetz, *Real-Time Systems : Design Principles for Distributed Embedded Applications*, 2nd ed. Springer Publishing Company, Incorporated, 2011.

[12] 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.

[13] A. Ghosal, T. Henzinger, C. Kirsch, and M. Sanvido, "Event-driven programming with logical execution times," in *Hybrid Systems : Computation and Control*, vol. 2993, 03 2004, pp. 357–371.

[14] D. Chabrol, G. Vidal-Naquet, V. David, C. Aussagues, and S. Louise, "Oasis : A chain of development for safety-critical embedded real-time systems," 01 2004.

[15] C. Baier and J.-P. Katoen, *Principles of model checking*. MIT press, 2008.

[16] A. Francalanza, L. Aceto, A. Achilleos, D. P. Attard, I. Cassar, D. Della Monica, and A. Ingólfsdóttir, "A foundation for runtime monitoring," in *International Conference on Runtime Verification*. Springer, 2017, pp. 8–29.

Effets des Régulateurs de Trafic dans les Réseaux en Temps Contraint

Ludovic Thomas
ISAE-SUPAERO
Université de Toulouse
ludovic.thomas@isae-supaero.fr

***Abstract*—Les réseaux en temps contraint sont utilisés dans les contextes critiques et doivent fournir des garanties de performances dans le pire cas, pas en moyenne. Le lissage par flux peut aider au calcul de garanties de bornes de latences. Le lissage peut être réalisé en utilisant des régulateurs par flux (comme le *Token Bucket Filter*, filtre du seau à jetons, dans Linux) ou les régulateurs entrelacés (comme avec IEEE TSN *Asynchronous Traffic Shaping*, lissage de trafic asynchrone). Au cours de la thèse, nous analysons les effets des régulateurs de trafic sur les bornes de performances des réseaux en temps contraint en utilisant la théorie du calcul réseau. En particulier, les régulateurs de trafic cassent le phénomène de cascade des bursts qui peut créer des dépendances cycliques qui peuvent à leur tour causer des pertes par congestion dans des réseaux pourtant sous-chargés. Nous développons ainsi un algorithme pour trouver, dans n'importe quel réseau, les positions optimales des régulateurs pour casser toutes les dépendances cycliques à coût minimal. Ensuite, nous observons que les propriétés des régulateurs reposent sur l'hypothèse que leur horloge interne est parfaite. Nous réévaluons les propriétés des régulateurs avec des modèles d'horloges réalistes et nous développons un modèle d'adversaire qui cause des pertes de congestion lorsque les régulateurs entrelacés sont utilisés (comme dans TSN ATS). Nous vérifions finalement le modèle d'adversaire sur des simulations avec ns-3. Au cours de la thèse, nous développons une boite à outils de résultats pour tenir compte des inexactitudes des horloges lors de l'utilisation du calcul réseau. Nous développons également FP-TFA, un algorithme basé sur l'analyse par flux total qui calcule des bornes de délais sur des réseaux avec ou sans dépendances cycliques, avec ou sans régulateurs.**

Motivations et contributions

Les réseaux en temps contraint supportent des applications temps-réel dans les domaines de l'avionique, l'espace, l'industrie et l'automobile. Le groupe de travail *time-sensitive networking* (réseaux en temps contraint) (TSN) de l'*Institute of Electrical and Electronics Engineers* (institut des ingénieurs en électricité et électronique) (IEEE) et le groupe de travail *deterministic networking* (réseaux déterministes) (DetNet) de l'*Internet Engineering Task Force* (groupe de travail pour l'ingénierie d'Internet) (IETF) visent à fournir des bornes déterministes sur le délai pire-cas.

Lisser les flux à l'intérieur du réseau à l'aide de régulateurs de trafic permet d'atteindre cet objectif: les régulateurs sont des composants placés avant un serveur qui suppriment l'augmentation de burst due à la compétition avec les autres flux dans les serveurs précédents. Ils permettent une plus grande efficacité des réseaux en temps contraint et existent en deux versions: les *per-flow regulators* (régulateurs par flux) (PFRs) et les *interleaved regulators* (régulateurs entrelacés) (IRs), la seconde version étant utilisée par TSN *asynchronous traffic shaping* (lissage de trafic asynchrone) (ATS).

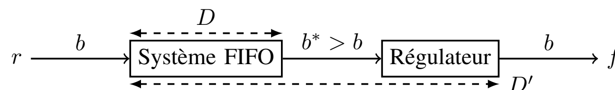


Fig. 1: Propriété de lissage gratuit. Le flux f avec un rate r et un burst initial b voit son burst augmenter à la sortie du système en raison de la contention avec les autres flux. Un régulateur est placé à la sortie pour forcer le flux f à présenter un burst de retour à b, en ralentissant les paquets si nécessaire. Si le système est *first in, first out* (premier arrivé, premier servi) (FIFO), le régulateur n'augmente pas le délai de pire cas (donc $D' = D$).

Lorsque l'horloge à l'intérieur d'un régulateur est parfaite, ce dernier jouit de la propriété de "lissage gratuit" [1] montrée en Figure 1. Plusieurs travaux reposent sur cette propriété pour étudier les combinaisons entre les régulateurs de trafic et les mécanismes d'ordonnancement par classe [2], [3]. Tous ces travaux supposent que les régulateurs sont déployés dans tous les nœuds du réseau. Cependant, les régulateurs sont coûteux: ils nécessitent de l'espace mémoire. Il serait intéressant d'être capable d'identifier, sur n'importe quel réseau, un petit nombre de positions décisives sur lesquelles l'installation d'un régulateur de trafic améliore le plus les performances garanties du réseau, tout en assurant sa stabilité.

Par ailleurs, l'horloge réellement utilisée par le régulateur n'est pas parfaite et dévie par rapport au temps exact. Elle peut être synchronisée avec une horloge maître, sa déviation est alors contenue par un protocole de synchronisation. Elle peut aussi être non-synchronisée. Le processus de standardisation de TSN ATS a soulevé des questions sur les conséquences possibles des imperfections des horloges lorsque les IRs sont utilisés dans un réseau.

Dans la thèse:

- Nous proposons *low-cost acyclic network* (réseau acyclique à faible coût) (LCAN), un algorithme qui trouve le nombre optimal de régulateurs pour casser toutes les dépendances cycliques dans un réseau.
- Nous proposons *FP-TFA*, un algorithme qui calcule des bornes de délai dans des réseaux avec ou sans dépendances cycliques et avec ou sans régulateurs.
- Nous proposons également un modèle d'horloge pour borner les imperfections des horloges. Nous fournissons un ensemble de résultats qui peuvent être combinés aux autres résultats de calcul réseau pour calculer des bornes de latences en présence d'horloges imparfaites.
- Dans les réseaux non-synchronisés, nous montrons que la configuration des régulateurs de trafic doit être adaptée.

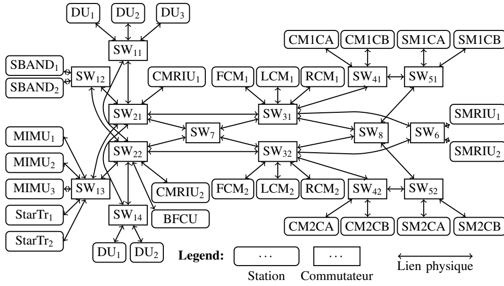


Fig. 2: Réseau avionique du vaisseau Orion de la NASA.

Sans cela, des pertes par congestion peuvent survenir. Pour les PFRs nous fournissons *asynchronous dual arrival-curve method* (méthode d'analyse asynchrone par double courbe d'arrivée) (ADAM), un algorithme calculant les paramètres de contrôle des régulateurs tout en conservant un plan de contrôle du réseau simple.

- Dans les réseaux synchronisés, nous montrons que les PFRs non-adaptés provoquent une pénalité de délai bornée. En revanche, quelle que soit la précision de synchronisation, il existe toujours une situation qui provoque des pertes de paquets dans les IRs qui ne sont pas adaptés. Nous fournissons un modèle d'adversaire décrivant les actions que l'adversaire opère pour déclencher cette instabilité. Ce modèle d'adversaire a été confirmé par des simulations sur ns-3.

Ludovic Thomas est encadré par Ahlem Mifdaoui, professeure à l'ISAE-SUPAERO, Toulouse, France et Jean-Yves Le Boudec, professeur à l'EPFL, Lausanne, Suisse. Une partie du travail présenté ici a déjà été publié dans [4], [5].

Nombre optimal de régulateurs avec LCAN

Prennons le réseau avionique du vaisseau d'exploration Orion de la NASA (Figure 2). Avec 119 flux multicast routés dessus, le réseau possède 300k dépendances cycliques [4]. Elles sont créées par le phénomène de cascade des bursts et elles restreignent la flexibilité du réseau.

En effet, si un outil de calcul réseau comme FP-TFA calcule des bornes de délais pour une configuration donnée avec une charge u du réseau ($u < 1$), alors une autre configuration avec une charge $u + \epsilon$ ($u + \epsilon < 1$) pourrait engendrer des délais non bornés et des pertes de paquets ! Pour limiter les conséquences d'un changement de configuration sur le réseau, nous pouvons retirer de ce dernier toutes les dépendances cycliques. Une fois cela fait, tant que la charge du réseau u reste inférieure à 1, le réseau restera stable et quelque soit la nouvelle configuration, des bornes de délais pourront être calculées avec des outils comme FP-TFA.

Les régulateurs de trafic cassent l'effet de cascade des bursts. Ainsi, placer des régulateurs à chaque serveur du réseau retire toutes les dépendances cycliques d'une manière évidente, mais cela vient avec un coût matériel élevé. Sur le réseau avionique Orion, cela nécessiterait 249 régulateurs. Pour réduire les coûts, notre outil *low-cost acyclic network* (réseau acyclique à faible coût) (LCAN) trouve le nombre optimal de régulateurs (et leurs positions) de telle sorte que toutes les dépendances cycliques soient cassées. Il se base sur une représentation du réseau avec des graphes, et chaque type de régulateurs est représenté par une opération particulière sur

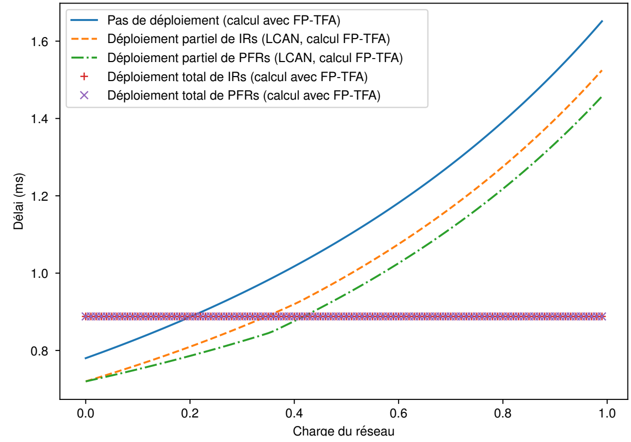


Fig. 3: Plus grande borne de délai parmi tous les flux dans le réseau avionique Orion, en fonction de la charge du réseau et avec différentes stratégies de placement des régulateurs dans le réseau. Les courbes les plus basses représentent les meilleures stratégies.

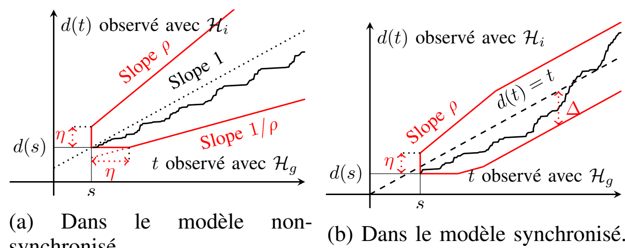
ces graphes. Le problème est ainsi ramené à un problème de *minimum feedback arc set* (ensemble minimal d'arcs de rétroaction) (MFAS) pour lequel des algorithmes optimaux efficaces existent. Sur le réseau Orion, LCAN supprime les dépendances cycliques en utilisant uniquement 8 PFRs ou 14 IRs et trouve la solution optimale en une dizaine de secondes.

Bornes de délais dans des réseaux quelconques avec ou sans régulateurs

Il nous faut à présent comparer les performances garanties de la proposition de placement de LCAN (déploiement partiel) avec celles de la méthode plaçant des régulateurs partout (déploiement total) et celles de la méthode n'en plaçant pas du tout (pas de déploiement). Pour cela, nous développons FP-TFA, un algorithme qui calcule des bornes de délai dans les réseaux avec ou sans dépendances cycliques et avec ou sans régulateurs.

FP-TFA se base sur les approches *total-flow analysis* (analyse par flux total) (TFA) et intègre les résultats de Mifdaoui et Leydier [6]. Ces derniers prennent en compte la sérialisation des données par le médium de transmission, ce qui réduit les bursts apparents des flux et permet d'obtenir des bornes de délai plus fines. FP-TFA intègre également des modèles pour PFR et IR de même qu'une méthode de point fixe qui permet de calculer des bornes de délais dans les réseaux sur lesquels les dépendances cycliques n'ont pas été retirées [4], [7].

Les résultats sur le réseau avionique d'Orion (tirés de [4]) sont présentés en Figure 3. Nous observons que le déploiement partiel des régulateurs avec LCAN fournit de meilleures bornes de délai par rapport à la méthode ne déployant aucun régulateur. Le gain est plus important avec PFR. En effet, l'IR ne peut supprimer les effets de contention que du nœud directement amont tandis que le PFR peut supprimer la contention cumulée sur plusieurs nœuds. La stratégie de déploiement total offre une borne de délai qui ne dépend pas de la charge du réseau. Ainsi, elle offre les meilleures bornes à forte charge, mais elle est moins intéressante à faible charge. Dans tous les cas, le déploiement partiel en utilisant LCAN offre un bon



(a) Dans le modèle non-synchronisé.

(b) Dans le modèle synchronisé.

Fig. 4: Enveloppe de $d_{g \to i}(t)$ (en rouge) et exemple d'évolution possible de $d_{g \to i}(t)$ (en noir black).

compromis entre les performances pire-cas et le coût matériel du déploiement des régulateurs.

Modélisation des effets des imperfections des horloges sur les performances pire-cas

Plus haut nous montrions que les régulateurs améliorent la flexibilité du réseau et qu'ils améliorent aussi ses performances pire-case en cas de forte charge, même lorsqu'ils ne sont que partiellement déployés. Ces observations reposent cependant sur la propriété de lissage gratuit, qui a été prouvée en utilisant le calcul réseau, mais uniquement lorsque le réseau utilise des horloges parfaites pour chaque composant.

De fait, la notion même de borne de délai dépend du modèle d'horloge utilisé pour la définir. Nous proposons un modèle d'horloge basé sur des imperfections réalistes des horloges. Il se base sur le modèle fourni dans [8] mais s'en distingue en bornant les évolutions relatives des horloges, plutôt qu'en se concentrant sur leurs propriétés stochastiques.

Pour une paire d'horloges $(\mathcal{H}_g, \mathcal{H}_i)$, nous notons $d_{g \to i}(t)$ l'heure que l'horloge $\mathcal{H}_i$ affiche lorsque l'horloge $\mathcal{H}_g$ affiche t. $d_{g \to i}$ est la fonction de temps relative de $\mathcal{H}_g$ à $\mathcal{H}_i$. Pour un réseau d'horloges non-synchronisées donné, nous définissons la borne sur la jigue temporelle η et la borne sur la stabilité des horloges ρ, tel que toute paire d'horloges $(\mathcal{H}_g, \mathcal{H}_i)$ dans le réseau vérifie $\forall t \geq s$

$$\frac{1}{\rho}(t - s - \eta) \leq d_{g \to i}(t) - d_{g \to i}(s) \leq \rho(t - s) + \eta \qquad (1)$$

Dans la Figure 4a, pour tout point de départ $(s, d(s))$, nous montrons l'espace d'évolution possible de $d(t)$ dans le modèle non-synchronisé, ainsi qu'une trajectoire acceptable. La fonction d'erreur de temps $t \mapsto d(t) - t$ peut être non-bornée dans ce modèle. η et ρ sont des paramètres du réseau et ne dépendent pas de la paire d'horloges considérée. Nous montrons que pour un réseau type TSN, nous pouvons prendre η =4ns et ρ =1+2·10^{-4} [5].

Si, de plus, les horloges du réseau sont synchronisées avec un protocole de synchronisation du temps, nous définissons la borne d'erreur de temps Δ telle que toute paire $(\mathcal{H}_g, \mathcal{H}_i)$, vérifie que $d_{g \to i}$ respecte les contraintes dans l'Équation (1), ainsi que:

$$\forall t, |d_{g \to i}(t) - t| \ \leq \Delta \qquad (2)$$

Dans la Figure 4b, pour tout point de départ $(s, d(s))$, nous présentons l'espace d'évolution possible de la fonction $d(t)$ dans le cadre du modèle synchronisé. Nous montrons de même une trajectoire acceptable. Notons que l'enveloppe de largeur Δ est centrée sur la droite de synchronisation parfaite $d(t) = t$ et non pas sur le point de départ.

TABLE I: Relation entre une courbe d'arrivée *leaky-bucket* (en seau percé) [resp. une courbe de service "rate et latence"] telle qu'observée avec $\mathcal{H}_i$ et une courbe d'arrivée [resp. de service] du même flux [resp. serveur] lorsqu'il est observé avec $\mathcal{H}_g$.

	Seau percé Courbe d'arrivée	Rate et latence Courbe de service
avec $\mathcal{H}_i$	$\gamma_{r,b}$	$\lambda_{R,T}$
avec $\mathcal{H}_g$, non-sync	$\gamma_{r\rho,b+r\eta}$	$\lambda_{R/\rho,\rho T+\eta}$
avec $\mathcal{H}_g$, sync	$\gamma_{r\rho,b+r\eta} \wedge \gamma_{r,b+2r\Delta}$	$\lambda_{R/\rho,\rho T+\eta} \vee \lambda_{R,T+2\Delta}$

La théorie de calcul réseau, utilisée pour obtenir des bornes de délais repose sur les fonctions cumulatives de type $A(t)$ qui compte le nombre total de bits observés à un point d'observation dans le réseau, jusqu'à l'instant t. Pour obtenir des bornes déterministes, le trafic des flux est supposé borné par une contrainte de courbe d'arrivée de la forme: $\forall t \geq s \geq 0, A(t) - A(s) \leq \alpha(t - s)$. Une courbe d'arrivée fréquemment utilisée est celle du seau percé $\gamma_{r,b}$ définie par $\gamma_{r,b}(t) = rt + b$ pour $t > 0$ et $\gamma_{r,b}(t) = 0$ pour $t \leq 0$. Elle correspond à un flux contraint par un rate r et un burst b.

Le service minimal offert par un serveur dans le réseau est également supposé borné par une condition de la forme $\forall t \geq 0 : D(t) \geq (A \otimes \beta)(t)$, avec A [resp. D] la fonction cumulative à l'entrée [resp. à la sortie] du serveur. La fonction β est alors une courbe de service du serveur et le symbole $\otimes$ représente la convolution dans l'algèbre min-plus [9]. De nombreux serveurs peuvent être modélisés par une contrainte de type "rate et latence" $\lambda_{R,T}(t) = \max(0, R(t - T))$, et un serveur FIFO qui garantie un délai maximal borné par D offre la courbe de service δ_D définie par $\delta_D(t) = 0$ pour $t \leq D$ et $\delta_D(t) = +\infty$ pour $t > D$. Les résultats de calcul réseau donnent des bornes de délai et de backlog à tout serveur pour lequel on connait la courbe de service ainsi que les courbes d'arrivée des flux entrants.

En utilisant notre modèle d'horloge et ses paramètres η, ρ et Δ (si synchronisé), nous prouvons un ensemble de résultats qui peuvent êtré utilisés pour obtenir une courbe d'arrivée d'un flux [resp. une courbe de service minimal d'un serveur], lorsqu'il est observé avec l'horloge $\mathcal{H}_g$ (et que l'on note alors $\alpha^{\mathcal{H}_g}$ [resp $\beta^{\mathcal{H}_g}$]) dès que l'on connaît une courbe d'arrivée [resp de service] de ce flux [resp ce serveur] lorsqu'il est observé par une horloge différente $\mathcal{H}_i$ (et que l'on note $\alpha^{\mathcal{H}_i}$ [resp $\beta^{\mathcal{H}_i}$]). Les résulats sont regroupés dans la Table I pour les courbes d'arrivée et de service les plus communes.

Conséquences des imperfections des horloges sur les régulateurs de trafic

Un PFR avec une horloge parfaite, configuré avec une courbe d'arrivée σ pour un flux f, s'assure que sa sortie respecte la contrainte de courbe d'arrivée σ (aussi nommée "courbe de lissage"). Si le flux f entre dans le PFR trop vite, ses paquets sont stockés dans le buffer du PFR (avec une file d'attente FIFO par flux) et ne sont relâchés que lorsque (et dès que) cela ne viole pas la contrainte de courbe d'arrivée. Ainsi, la mesure du temps écoulé est au cœur du fonctionnement du PFR. Un IR est similaire à un PFR mais tous les paquets de tous les flux lissés sont stockées dans une unique file d'attente FIFO. Le paquet en tête de la file est relâché dès que cela ne viole pas la contrainte de courbe d'arrivée du flux auquel le

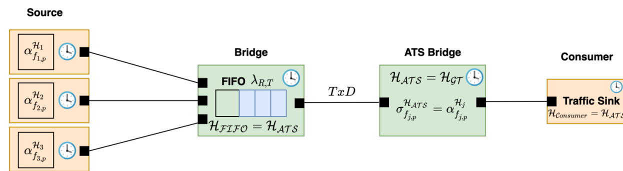


Fig. 5: Simulation de l'instabilité due aux imperfections des horloges avec un IR: trois sources génèrent du trafic dans un serveur, à la sortie du serveur un IR lisse les flux. L'adversaire contrôle les horloges des trois sources.

paquet appartient. Tous les autres paquets, y compris d'autres flux, doivent attendre d'arriver à la tête de la file d'attente avant d'être considérés par le mécanisme de lissage.

En réalité, la courbe de lissage imposée par un régulateur n'est pas exactement σ. Dans un réseau non synchronisé, les deux types de régulateurs doivent être adaptés: leur courbe de lissage σ doit prendre en compte les paramètres du modèle d'horloge ρ, η. Dans le cas contraire, ils peuvent générer des délais non bornés ou des pertes par congestion. Nous détaillons deux méthodes: "cascade de rate et de burst" et *asynchronous dual arrival-curve method* (méthode d'analyse asynchrone par double courbe d'arrivée) (ADAM) et nous montrons qu'elles n'augmentent que très légèrement les bornes de délai par rapport à la situation hypothétique avec des horloges parfaites. ADAM nécessite un plan de contrôle plus simple que la première méthode car tous les régulateurs sur le chemin du flux sont alors configurés avec les même courbes de lissage pour ce flux.

Pour un réseau synchronisé, nous exhibons une différence fondamentale: la pénalité d'un PFR non-adapté est bornée et dépend de la précision de synchronisation, tandis que pour un IR, quelle que soit la précision de la synchronisation, un IR non-adapté peut engendrer des délais non bornés et des pertes de congestion.

Pour montrer cette instabilité, nous développons un modèle d'adversaire. Dans ce modèle, l'ingénieur choisit les paramètres du modèle de synchronisation $\rho > 1$, $\Delta > 0$ et $\eta \geq 0$. L'adversaire contrôle certaines horloges du réseau (au moins trois horloges) et peut accélérer ou ralentir les horloges quand il le souhaite, à condition de rester dans l'espace d'évolution autorisé de la Figure 4b. Rappelons que la taille de cette enveloppe rouge est contrôlée par l'ingénieur et donc l'espace d'évolution accessible à l'adversaire peut être aussi petit que l'ingénieur le souhaite. Quels que soient les choix de l'ingénieur, l'adversaire peut toujours créer une instabilité en adoptant la stratégie décrite dans [5].

Ce modèle d'adversaire a été implémenté dans le simulateur de réseau ns-3 [10] par Guillermo Aguirre dans le cadre d'un projet de master à l'EPFL sous la supervision du Prof. Le Boudec et avec l'aide de Ludovic Thomas. La Figure 5 montre une simulation réalisée sur ns-3 avec trois sources, un serveur (le *Bridge*) et un IR (le *ATS Bridge*). La Figure 6 présente le délai de bout en bout en fonction du numéro du paquet: on observe qu'il croit linéairement (alors que le réseau est sous-chargé !) et dépasse au bout d'un moment la borne de performance calculée avec le calcul réseau lorsque les horloges sont parfaites.

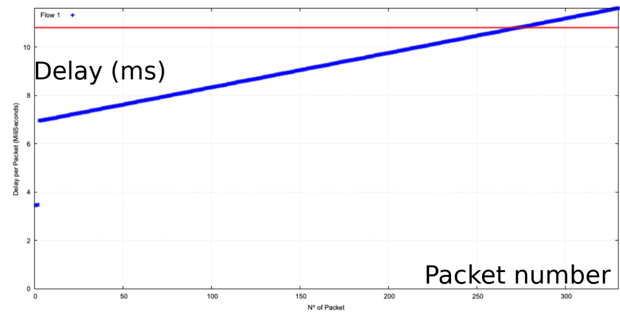


Fig. 6: Résultats de la simulation: délai de bout en bout par paquet (en bleu). La ligne rouge est la borne de délai calculée lorsque les horloges sont parfaites. L'adversaire réussi à produire des délais de bout en bout qui dépassent cette borne.

Conclusion

La thèse étudie les propriétés des régulateurs de trafic sur les performances déterministes des réseaux en temps contraint. Des algorithmes de calcul de bornes de délai (FP-TFA) et de placement idéal des régulateurs (LCAN) sont développés. L'hypothèse d'horloge parfaite est remise en question. Un modèle d'horloge est proposé pour le calcul réseau et l'impact des imperfections sur les bornes de délai est étudiée.

References

[1] J.-Y. Le Boudec, "A Theory of Traffic Regulators for Deterministic Networks With Application to Interleaved Regulators," *IEEE/ACM Transactions on Networking*, vol. 26, no. 6, pp. 2721–2733, Dec. 2018, http://doi.org/10.1109/TNET.2018.2875191.

[2] L. Zhao, P. Pop, and S. Steinhorst, "Quantitative Performance Comparison of Various Traffic Shapers in Time-Sensitive Networking," *arXiv:2103.13424 [cs]*, Mar. 2021, http://arxiv.org/abs/2103.13424.

[3] E. Mohammadpour, E. Stai, M. Mohiuddin, and J. Le Boudec, "Latency and Backlog Bounds in Time-Sensitive Networking with Credit Based Shapers and Asynchronous Traffic Shaping," in *2018 30th International Teletraffic Congress (ITC 30)*, vol. 02, Sep. 2018, pp. 1–6, http://doi.org/10.1109/ITC30.2018.10053.

[4] L. Thomas, J.-Y. Le Boudec, and A. Mifdaoui, "On Cyclic Dependencies and Regulators in Time-Sensitive Networks," in *2019 IEEE Real-Time Systems Symposium (RTSS)*, Dec. 2019, pp. 299–311.

[5] L. Thomas and J.-Y. Le Boudec, "On Time Synchronization Issues in Time-Sensitive Networks with Regulators and Nonideal Clocks," *Proceedings of the ACM on Measurement and Analysis of Computing Systems*, vol. 4, no. 2, pp. 27:1–27:41, Jun. 2020, https://doi.org/10.1145/3392145.

[6] A. Mifdaoui and T. Leydier, "Beyond the Accuracy-Complexity Trade-offs of Compositional Analyses using Network Calculus for Complex Networks," in *10th International Workshop on Compositional Theory and Technology for Real-Time Embedded Systems (Co-Located with RTSS 2017)*, Paris, France, Dec. 2017, pp. pp. 1–8, https://hal.archives-ouvertes.fr/hal-01690096.

[7] A. Bouillard, M. Boyer, and E. Corronc, *Deterministic Network Calculus: From Theory to Practical Implementation*, ser. Networks and Telecommunications. Wiley, 2018, http://doi.org/10.1002/9781119440284.

[8] ITU, "Definitions and terminology for synchronization networks," *ITU G.810*, 1996, https://www.itu.int/rec/T-REC-G.810-199608-I/en.

[9] J.-Y. Le Boudec and P. Thiran, *Network Calculus: A Theory of Deterministic Queuing Systems for the Internet*, ser. Lecture Notes in Computer Science, Lect.Notes Computer. Tutorial. Berlin Heidelberg: Springer-Verlag, 2001, https://www.springer.com/us/book/9783540421849.

[10] nsnam, "Ns-3 Network Simulator. Project homepage," https://www.nsnam.org/, 2011.

Une approche de génération automatique de configuration basée sur les modèles pour les réseaux TSN

Maxime Samson*†, Thomas Vergnaud*, Éric Dujardin*, Laurent Ciarletta† and Ye-Qiong Song†
* Thales Research & Technology – Palaiseau, France
{maxime.samson – eric.dujardin – thomas.vergnaud}@thalesgroup.com
† LORIA – Université de Lorraine – Nancy, France
{maxime.samson – ye-qiong.song – laurent.ciarletta}@loria.fr

***Abstract*—Les nouvelles fonctionnalités qu'apportent les standards définis par le groupe de travail IEEE 802.1 TSN à la commutation Ethernet permettent son utilisation pour des réseaux temps réel. Ces nouvelles fonctionnalités rendent possible la conception de réseaux Ethernet déterministes, mais au prix d'un effort de configuration très important. Cette difficulté de configuration s'applique à la fois aux équipements réseaux et aux outils utilisés pour concevoir ces réseaux, par exemple des simulateurs.**

Dans cet article, nous présentons une approche basée sur les modèles qui permet la génération automatique de la configuration d'un réseau TSN pour des outils tels que des simulateurs. Cette approche simplifie l'étape de configuration réseau et assure la cohérence entre les configurations générées pour les différentes cibles.

I. Introduction

Time Sensitive Networking (TSN) est un ensemble de standards développés par le groupe de travail IEEE 802.1 TSN, qui était anciennement nommé Audio/Video Bridging (AVB).

Cet ensemble de standards a pour objectif de compléter le standard 802.1Q [1] afin de rendre possible l'utilisation d'Ethernet pour des communications déterministes.

Pour ce faire, les standards TSN apportent un ensemble de nouvelles fonctionnalités, p.ex. une notion de temps commune au réseau établie par synchronisation temporelle, la réservation des ressources pour les flux de données, la régulation des flux.

Un réseau Ethernet déterministe permet des communications à criticité mixte avec des garanties de qualité de service et une bande passante élevée, ce qui intéresse des secteurs comme l'automatisation industrielle, l'automobile et l'aviation.

Dans le secteur automobile par exemple, les réseaux TSN présentent plusieurs avantages. TSN permet des communications de criticité différentes. Il est donc possible de n'utiliser qu'un seul réseau pour les communications critiques, qui utilisent généralement un bus Controller Area Network (CAN), et un autre réseau (p.ex. MOST) pour les communications audio et vidéo liées au divertissement [2]. L'utilisation d'un réseau unique permet notamment une réduction du câblage et donc du poids des véhicules.

Les nouvelles fonctionnalités apportées par les standards TSN entraînent une augmentation de la complexité de la configuration qu'il faut déployer sur le réseau [3], [4].

Cela s'applique également à l'utilisation de simulateurs ou d'émulateurs réseau, ce qui est courant lors de la phase de conception. Ce problème vient s'ajouter à un problème existant lié à l'utilisation de ces outils : la nécessité de savoir configurer chacun des outils que l'on souhaite utiliser.

L'approche basée sur la génération automatique de configuration à partir de modèles que nous présentons dans cet article permet de résoudre ces problèmes. La génération de la configuration permet tout d'abord d'éviter les erreurs humaines. Elle permet également de ne pas être obligé de savoir configurer chacun des outils utilisés lors de la phase de conception. Cette approche permet donc un gain de temps important.

Des travaux sur la génération automatique de configuration pour les réseaux TSN existent déjà. Dans [5], TSNSched, un outil de génération de configuration pour un mécanisme de régulation de flux de TSN, le Time Aware Shaper (TAS), est présenté. Cet outil utilise un démonstrateur automatique et un système de contraintes. L'approche de modélisation de réseaux qui est présentée n'est pas découplée de l'outil de génération, ce qui ne la rend pas exploitable par d'autres outils. La limite principale de ces travaux est que la configuration générée ne concerne que le TAS est n'est donc pas exploitable par un outil de conception comme un simulateur.

L'outil présenté dans [4] permet de générer automatiquement la configuration d'un réseau TSN pour un simulateur réseau. Cet outil prend la forme d'une extension du simulateur et lui est donc liée. Il n'est donc pas possible d'utiliser cet outil pour générer de la configuration pour différents outils de conception ou pour du matériel réel.

Dans la suite de cet article, nous commencerons par expliquer plus en détail le problème posé par la complexité de la configuration d'un réseau TSN puis nous présenterons la solution que nous proposons et son implémentation.

II. TSN

Les standards TSN définissent plusieurs mécanismes de régulation de flux, dont certains sont basés sur la division temporelle, qui permettent de garantir le respect des contraintes temps réel des flux de données. La synchronisation temporelle de tous les nœuds du réseau est un mécanisme de base de TSN. Cette synchronisation est assurée par le standard IEEE 802.1AS qui définit le Generic Precision Time Protocol (gPTP).

Afin d'assurer que le réseau sera bien capable de respecter les contraintes temps réel des différents flux de données, les ressources nécessaires doivent être réservées au niveau des commutateurs. Par exemple, la somme des bandes passantes nécessaires aux flux transmis par un port d'un commutateur doit être inférieure à la capacité totale de ce port.

Un des mécanismes de régulation de flux de TSN est le Credit-Based Shaper (CBS). Il lisse le trafic, ce qui empêche des gros flux de données de saturer temporairement le réseau. Pour cela, il définit des classes de trafic, et l'utilisateur leur attribue le taux auquel elles gagneront des crédits. Les trames appartenant à ces classes de trafic ne sont transmises que si la quantité de crédits de leur classe est positive ou nulle. Quand des trames sont émises, leur classe de trafic perd des crédits, et elle en gagne pendant qu'elles sont en attente de transmission.

L'utilisation du CBS implique nécessairement des périodes pendant lesquelles les trames des classes de trafic qu'il régule seront en attente. Ce comportement n'est pas souhaitable pour la régulation des flux de données les plus critiques, pour lesquels la latence doit être la plus basse possible. Pour atteindre cet objectif, le standard IEEE 802.1Qbv définit le Time Aware Shaper (TAS). Le rôle de ce mécanisme est de répartir l'accès aux liens du réseau dans le temps afin d'isoler les flux les plus critiques et de leur garantir une latence et une gigue minimale. Le TAS définit, au niveau des ports des commutateurs, la durée d'un cycle puis divise ce cycle en plusieurs fenêtres pendant lesquelles seules certaines classes de trafic pourront transmettre. Cette division temporelle est présentée dans la figure 1 sous la forme d'une table qui contient trois fenêtres temporelles. La sélection finale de la prochaine trame à transmettre est faite en respectant le code de priorité, présent pour chaque trame dans l'entête défini par le standard IEEE 802.1Q.

La figure 1 présente les mécanismes de régulation de flux utilisés par les ports de sortie des commutateurs d'un réseau TSN. On voit que les nœuds émetteurs des flux de données suivent chacun leur ordonnancement; en fait, chaque port de sortie des commutateurs peut être configuré de façon particulière.

Chaque port de chaque commutateur susceptible de transmettre un flux de données doit être configuré individuellement et ces configurations doivent permettre de respecter les contraintes de chaque flux. Cette multiplicité des configurations est ce qui crée la complexité de la configuration globale d'un réseau TSN, qui vient s'ajouter aux problématiques classiques, comme le routage.

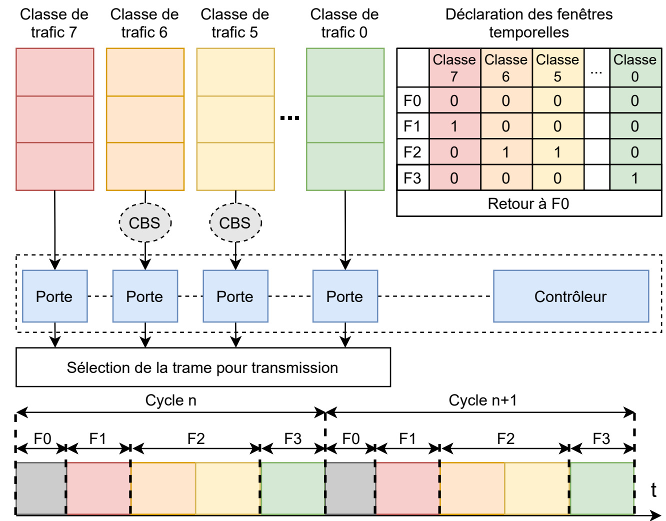


Fig. 1. Représentation des mécanismes utilisés par un port de sortie d'un commutateur TSN

III. Solution

A. Présentation de la solution

Afin de résoudre les problèmes que pose la complexité de la configuration d'un réseau TSN, notre approche comporte plusieurs étapes, présentées dans la figure 2.

La première étape est l'expression des contraintes que le réseau doit être capable de respecter. Ces contraintes ne suivent pas de formalisme particulier et peuvent concerner plusieurs aspects du réseau.

Les contraintes les plus importantes sont celles qui concernent les différents flux de données qui doivent circuler sur le réseau. Elles doivent tout d'abord permettre de dimensionner les flux. Par exemple, dans le cas de flux périodiques, elles expriment la taille des données et la période de transmission. Elles doivent ensuite définir la latence que le flux ne doit jamais dépasser.

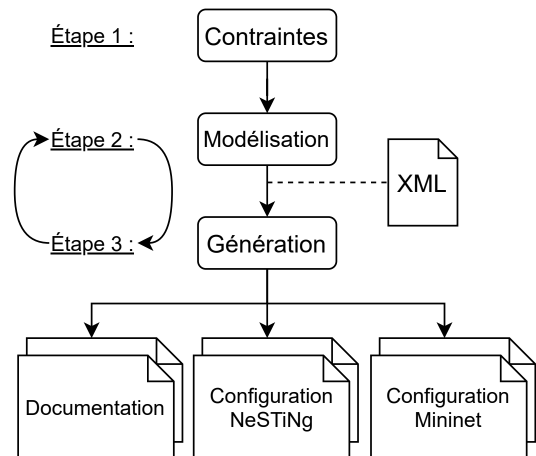


Fig. 2. Présentation de l'approche

La topologie du réseau peut également être contrainte. Le réseau peut être soumis à des problématiques d'espace ou de poids comme dans une voiture ou sur un drone. Les équipements peuvent en être une autre source, par exemple le nombre de commutateurs à utiliser ou le nombre de ports que possède chacun d'entre eux.

La deuxième étape de notre approche est la modélisation. L'objectif de cette étape est de créer une représentation de chaque élément du réseau qui respecte les contraintes exprimées à l'étape précédente et qui contienne les informations nécessaires à la génération automatique de configuration. Le modèle ainsi créé servira ensuite à assurer la cohérence entre les différentes configurations générées.

Enfin, la dernière étape est la génération de la configuration. Cette étape se base sur le modèle créé à l'étape précédente et la configuration est générée pour les cibles spécifiées par l'utilisateur.

Pour assister la conception du réseau, il est possible d'itérer sur les étapes 2 et 3 en évaluant le comportement du réseau dans un simulateur pour faire évoluer le modèle du réseau, et générer à nouveau une configuration.

B. *Mise en œuvre de la solution*

1) Modélisation: Notre modèle de déploiement est inspiré des concepts du standard MARTE [6], et plus particulièrement du chapitre GRM – Generic Resource Modeling. Nous nous reposons sur deux concepts qui y sont définis: ComputingResource et CommunicationMedia.

Nous reprenons le concept de ComputingResource pour décrire un nœud de calcul au sens large. Dans notre cas, nous l'utilisons pour représenter une machine ou un commutateur TSN. À partir de CommunicationMedia, nous décrivons les bus au sens large. Dans notre cas, nous l'utilisons pour modéliser les liens Ethernet et les flux.

Nous n'utilisons donc pas l'intégralité de ce qu'offre le standard MARTE mais nous nous en inspirons en l'adaptant. MARTE est un standard bien connu du domaine des systèmes embarqués temps réel, ce statut fait de lui un bon point de départ pour la modélisation de réseaux Ethernet déterministes.

Notre approche repose sur la notion de *ressource*: nous utilisons les termes ComputationResource (compRsc) et CommunicationResource (commRsc).

Nous faisons la distinction entre définition (compRscDef et commRscDef) et instanciation (compRsc et commRsc). La définition d'un commutateur permet d'expliciter l'ensemble des paramètres nécessaire à sa caractérisation. L'instanciation d'un commutateur consiste à créer un exemplaire de commutateur faisant référence à la définition; les valeurs des paramètres sont spécifiées dans l'instanciation. Le listing 1 montre la définition d'un flux de données.

```
<commRscDef name="Stream">
  <configParam max="1" min="0" name="deadline"
               type="time_t"/>
  <configParam max="1" min="1" name="payload"
               type="payload_t"/>
  <configParam max="1" min="1" name="pcp"
               type="pcp_t"/>
  <configParam max="1" min="1" name="vlan_id"
               type="vlan_id_t"/>
  <rscParam max="1" min="1" name="talker">
    <allowedRscDef ref="Ethernet_Interface"/>
  </rscParam>
  <structParam max="-1" min="1"
               name="listeners">
    <rscParam max="1" min="1"
              name="listener_port">
      <allowedRscDef
              ref="Ethernet_Interface"/>
    </rscParam>
    <structParam max="-1" min="1"
                 name="paths">
      <rscParam max="-1" min="1"
                name="path_member">
        <allowedRscDef
                ref="Ethernet_Interface"/>
      </rscParam>
    </structParam>
  </structParam>
</commRscDef>
```

Listing 1. Définition d'un flux de données

Le listing 2 montre l'instanciation d'un flux de données conforme à cette définition. Ce flux de données appartient à la classe de trafic la plus prioritaire et son délai de bout en bout ne doit pas dépasser 2 millisecondes. C'est un flux périodique dont les trames sont émisent toutes les 250 microsecondes et contiennent 128 octets. Il est émis par *talker1*, a pour seul destinataire *listener1* et doit traverser *switch1* et *switch2*.

```
<commRsc def="Periodic_Stream" name="stream1">
  <config def="deadline" value="{2, ms}"/>
  <config def="payload" value="{128, B}"/>
  <config def="pcp" value="7"/>
  <config def="period" value="{250, us}"/>
  <config def="vlan_id" value="1"/>
  <rscConfig def="talker"
           value="talker1_eth0"/>
  <structConfig def="listeners">
    <rscConfig def="listener_port"
             value="listener1_eth0"/>
    <structConfig def="paths">
      <rscConfig def="path_member"
               value="switch1_eth0"/>
      <rscConfig def="path_member"
               value="switch2_eth2"/>
    </structConfig>
  </structConfig>
</commRsc>
```

Listing 2. Instanciation d'un flux de données

Notre travail a consisté à établir les définitions des concepts, tels le commutateur ou le flux de donnée. Le travail de l'utilisateur consiste alors à instancier ces concepts. Il sait ainsi quelles informations il doit fournir pour chaque instanciation, ce qui le guide dans la spécification du réseau.

2) Génération: Notre outil de génération de configuration s'appuie sur les modèles d'instanciation établis au format XML lors de l'étape précédente. L'utilisateur spécifie le modèle à utiliser et les cibles pour lesquelles la configuration doit être générée.

Après avoir extrait les données du modèle, l'outil génère un ensemble de fichiers contenant de la documentation sur le modèle utilisé et les fichiers de configurations. Les cibles pour lesquelles la configuration peut être générée sont, pour l'instant, Mininet [7] et NeSTiNg [8].

Mininet est un émulateur réseau conçu pour SDN (Software Defined Network) permettant de créer un réseau composé de commutateurs, de liens et de terminaux virtuels. Un réseau virtuel permet, sur une seule machine, de prototyper, de tester et de déboguer un réseau avant de le déployer. Il est possible de spécifier la topologie à utiliser grâce à une interface de programmation Python. Cette interface permet de facilement effectuer des modifications sur la topologie et donc de tester différentes configuration du réseau.

NeSTiNg est un modèle de simulation de réseau TSN qui repose sur OMNeT++[1] et son framework INET[2]. Il met à disposition de ses utilisateurs des éléments permettant de simuler des réseaux TSN dans OMNeT++. Ces éléments incluent des commutateurs et des terminaux qui possèdent les fonctionnalités spécifiques à TSN.

La configuration d'une simulation utilisant NeSTiNg se fait par le biais de plusieurs fichiers : un fichier contenant la description de la topologie, un fichier contenant les paramètres de la simulation et plusieurs fichiers XML décrivant les flux de données, le routage et la configuration du TAS. Ces fichiers suivent tous une syntaxe stricte et peuvent donc être générés automatiquement.

La génération de la topologie dans les deux formats (Mininet et NeSTiNg) à partir d'un unique modèle permet de conserver la cohérence de la configuration.

Le listing 3 montre l'initialisation d'un flux de données dans NeSTiNg.

```
talker1.numApps = 1
talker1.app[0].typename =
    ↪ "UdpScheduledTrafficApp"
talker1.app[0].trafficGenerator.localPort =
    ↪ 1000
talker1.app[0].scheduleManager.
    ↪ initialAdminSchedule = xmldoc("xml/flows
    ↪ .xml", "/schedules/datagramSchedule[@id
    ↪ ='0']")

listener1.numApps = 1
listener1.app[0].typename = "UdpSink"
listener1.app[0].localPort = 1000
```

Listing 3. Initialisation d'un flux de données dans NeSTiNg

Le listing 4 montre l'instanciation d'un flux de données dans NeSTiNg à laquelle l'initialisation fait référence.

```
<datagramSchedule id="0" cycleTime="250us">
  <event payloadSize="128B"
         destAddress="listener1"
         destPort="1000" pcp="7" vid="1"/>
</datagramSchedule>
```

Listing 4. Instanciation d'un flux de données dans NeSTiNg

[1]https://omnetpp.org/

[2]https://inet.omnetpp.org/

Pour le modèle d'un réseau composé de 3 commutateurs et de 6 terminaux, l'outil de génération de configuration utilisé en spécifiant NeSTiNg comme cible génère environ 650 lignes dont environ 400 lignes de configuration pour NeSTiNg. Le temps d'exécution de la génération est en moyenne, sur 20 exécutions, de 123 ms pour une machine sous openSUSE Leap 15.2 équipée d'un Core i7 6820HQ.

IV. Conclusion

Dans cet article, nous avons présenté une approche de génération automatique de configuration basée sur des modèles. Cette approche consiste à réaliser un modèle du réseau pour lequel l'utilisateur veut générer la configuration, en respectant les contraintes définies en amont. Chaque élément d'un réseau, p.ex. les commutateurs ou les terminaux, possède sa propre définition. Le modèle doit respecter ces définitions et contenir les données nécessaires à l'instanciation de chacun de ces éléments. Le modèle s'exprime dans une syntaxe XML simple. Notre générateur de configuration génère alors la configuration pour la cible choisie.

Notre approche présente l'avantage de faciliter l'utilisation de différents outils lors de la phase de conception d'un réseau. Elle évite à l'utilisateur d'avoir à maîtriser la configuration de chacun de ces outils et elle assure la cohérence entre les différentes configurations puisqu'elles ont toutes été générées à partir du même modèle.

Nous avons prévu comme travaux futurs d'améliorer le générateur de configuration. Nous envisageons d'ajouter une fonctionnalité permettant de compléter des modèles. Cela permettra d'alléger la charge de l'utilisateur lors de l'étape de modélisation, notamment en calculant les durées des cycles TAS et des fenêtres temporelles, en s'appuyant, par exemple, sur un outil comme TSNSched [5]. Nous prévoyons également d'ajouter la possibilité de générer la configuration pour du matériel réel, par exemple avec un modèle YANG.

References

[1] *IEEE 802.1Q 2018 – Bridges and Bridged Networks*, IEEE Std., 2018.

[2] N. Navet, Y. Song, F. Simonot-Lion, and C. Wilwert, "Trends in automotive communication systems," *Proc. IEEE*, vol. 93, no. 6, pp. 1204–1223, 2005.

[3] W. Steiner, S. S. Craciunas, and R. S. Oliver, "Traffic planning for time-sensitive communication," *IEEE Communications Standards Magazine*, vol. 2, no. 2, pp. 42–47, 2018.

[4] B. Houtan, A. Bergström, M. Ashjaei, M. Daneshtalab, M. Sjödin, and S. Mubeen, "An automated configuration framework for tsn networks," in *22nd IEEE International Conference on Industrial Technology (ICIT'21)*, March 2021.

[5] A. C. T. d. Santos, B. Schneider, and V. Nigam, "Tsnsched: Automated schedule generation for time sensitive networking," in *2019 Formal Methods in Computer Aided Design (FMCAD)*, 2019, pp. 69–77.

[6] *UML Profile for MARTE: Modeling and Analysis of Real-Time Embedded Systems*, OMG Std., 2019.

[7] B. Lantz, B. Heller, and N. McKeown, "A network in a laptop: rapid prototyping for software-defined networks," in *Proceedings of the Ninth ACM SIGCOMM Workshop on Hot Topics in Networks - Hotnets '10*. ACM Press, pp. 1–6.

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

Towards robust network synchronization with IEEE 802.1AS

Quentin Bailleul
IRT Saint Exupéry
Toulouse, France
quentin.bailleul@irt-saintexupery.com

Katia Jaffrès-Runser, Jean-Luc Scharbarg
IRIT, Université de Toulouse, Toulouse INP
Toulouse, France
{katia.jaffres-runser, jean-luc.scharbarg}
@irit.fr

Philippe Cuenot
Continental
Toulouse, France
philippe.cuenot@continental-corporation.com

***Abstract*—IEEE 802.1AS is the synchronization protocol associated to the novel real-time switched Ethernet solution called Time Sensitive Networking (TSN). We discuss and provide first insights on the design choices required for a critical embedded network context. More specifically, our aim is to extract the key features that offer both a target synchronization precision and a robust setting to mitigate the loss of synchronization.**

I. INTRODUCTION

Fieldbuses cannot cope with the increasing communication needs of current embedded systems that aim at supporting a diverse set of flows, some of them carrying video and necessitating thus a large bandwidth. As such, real-time Ethernet solutions have been designed to bring much higher bandwidth and advanced quality of service policies. The challenge is then to guarantee that time sensitive flows can meet very low latency constraints in a network composed of multiple Ethernet switches. Moreover, some embedded distributed applications may require precise timestamping. For all these reasons, synchronization has become a central service in real-time switched Ethernet networks. This paper investigates the IEEE 802.1AS protocol [1], proposed by the IEEE TSN working group. Synchronization is required for the implementation of the real-time TSN shapers such as the Time-Aware Shaper (TAS) or the Cyclic Queuing and Forwarding (CQF) one.

The IEEE 802.1AS protocol is a profile of the IEEE 1588 [2] synchronization standard already in use in non-critical systems. IEEE 802.1AS has been designed with the goal of reaching a precision of less than 1 microsecond in a linear network setting where a master clock and an ordinary clock are separated by 7 hops. Simulation and worst case analysis [3] but also experimental measurements [4] have assessed its performance. However, for a standard to be adopted in a critical network, it is necessary to study the behavior of the protocol in the event of a failure.

In this paper, we address the problem of designing a robust and precise version of 802.1AS. First, we underline how robustness can be enforced with this standard. Second, we analyze what impacts the quality of synchronization using simulations to guide the design of a robust and precise deployment methodology for IEEE 802.1AS.

II. IEEE 802.1AS OVERVIEW

IEEE 802.1AS [1] is a profile of IEEE 1588 Precision Timing Protocol [2] for Time Sensitive Networking. Sometimes called generalized Precision Time Protocol (gPTP), this protocol is used to synchronize clocks across a network using the master slave paradigm. Each port of a time-aware system has one of the following states:

- Master : sends time synchronization information to the slave port of a time-aware system located at the other end of the physical link.
- Slave : receives time synchronization information from the master port.
- Passive : ignores time synchronization information to avoid loops.

The time-aware system with all its ports in master state is called Grandmaster and it is the time source of the network. It can be synchronized by an external time source (GPS, NTP, ...) or use its internal clock.

To determine port states, IEEE 802.1AS provides two methods. The first method is the external port state configuration. This method allows to statically define the state of the ports for all the devices involved in the synchronization. The second method is the Best Master Clock Algorithm (BMCA). This distributed algorithm is executed on each time-aware system to eventually determine the state of the local ports and to elect the Grandmaster by comparing the information received from each Grandmaster candidate.

Synchronization itself is based on two core mechanisms: *i)* the distribution of synchronization information and *ii)* the measurement of the link delay using the peer-to-peer delay mechanism.

The distribution mechanism is based on the transmission of `Sync` and `Follow_Up` messages that allow each time-aware system to synchronize to the Grandmaster clock. Every synchronization interval, which is typically of 125ms, the Grandmaster sends a `Sync` message out of its master ports, followed by a `Follow_Up` message containing t_0, the exact sending time of the `Sync` message, as pictured in Fig. 1. These two messages are received via the slave ports of the equipment connected to the Grandmaster. If the receiving device has at least one port in the master state then it will forward the `Sync` message to the next time-aware system. It forwards as well the `Follow_Up` message that carries t_0 and the data $T_{prop}+T_{res}$. Here, T_{prop} is the propagation delay measured with the other peer-to-peer delay mechanism and T_{res} is the residence time of the `Sync` message in the switch. These operations are repeated at each hop, until the complete set of network equipment is reached.

The peer-to-peer delay mechanism defined in IEEE 802.1AS measures the link propagation delay. It consists in an

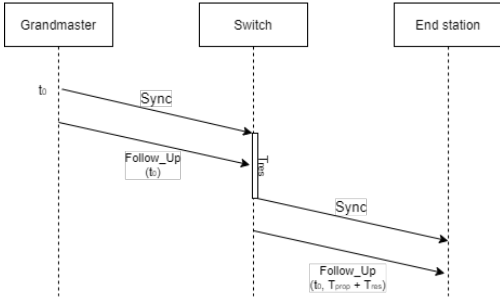


Fig. 1: Distribution mechanism.

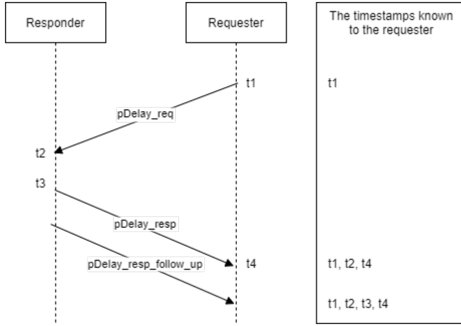


Fig. 2: Peer to peer delay mechanism.

exchange of `Pdelay` messages carrying timestamps between two time-aware systems that are separated by one hop. This mechanism is executed every *Pdelay interval* which is of 1s by default. To measure this propagation delay, the protocol needs four timestamps as depicted in Fig. 2. The first timestamp $t1$ is measured when the `Pdelay_req` is issued. Timestamp $t2$ is obtained upon receipt of this message. Timestamp $t3$ is measured when the `Pdelay_resp` is sent. Finally, $t4$ is measured upon reception of `Pdelay_resp`. With the formula (1), the mechanism can calculate the delay of the link, T_{prop}, from the timestamps. Moreover, with the $t3$ and $t4$ timestamps of two consecutive Pdelay procedures, the requester can extract the neighbor rate ratio nr of Eq. (2) in order to compensate the relative clock drift in Eq. (1).

$$T_{prop} = \frac{(t2 - t3) + nr.(t4 - t1)}{2} \quad (1)$$

Equation (1) makes the hypothesis of the symmetry of the link but the protocol makes it possible to compensate for the existing asymmetries if they can be estimated, typically in a calibration step.

Using the two mechanisms described above, each device can adjust its local clock. Indeed, to deduce the current time, the device just sets its clock to t_0, the original time of transmission of the `Sync`, and adds the previous propagation delay and residence times accumulated in the `Follow_Up` message and the propagation delay of the last hop T_{prop} measured with the peer-to-peer delay mechanism.

$$nr = \frac{f_{req}}{f_{resp}} = \frac{t3_i - t3_{i-1}}{t4_i - t4_{i-1}} \quad (2)$$

In the 2020 version of IEEE 802.1AS [5], a domain mechanism has been added. A domain is made up of several time-aware systems participating in the synchronization including a Grandmaster. A time-aware system can belong to several domains, originating from the same Grandmaster or from a different one. In the latter case, the backup Grandmaster is called the *hot-standby* Grandmaster. It is important to note that all these domains are active together at the same time, with proper `Sync` and `Follow_Up` messages sent in the network. However, the function and thus the criteria that trigger a node to adopt one of the domains it is currently synchronized with has not yet been standardized. Defining this function is one of the objectives of the upcoming IEEE 802.1ASdm amendment.

III. A CONFIGURATION FOR ROBUST SYNCHRONIZATION

For a critical on-board network, ensuring robustness is compulsory for the deployment of IEEE802.1AS. The BMCA offers robustness in the event of a link, time-aware system or Grandmaster failure by re-configuring the state of the ports to allow synchronization information to be broadcast again. This is a very flexible mechanism, but for which it may be difficult to predict which spanning tree will be used for the dissemination of `Sync` and `Follow_Up` messages at any time. In addition, the reconfiguration time may depend on the order of arrival of the messages describing the quality of a Grandmaster candidate and reconfiguration may take up to several seconds.

Nonetheless, the use of multiple domains combined with the external configuration of the port state makes it possible to obtain robustness under certain conditions. Indeed, it is necessary that the spanning trees used to distribute synchronization information in the different domains offer robustness to time-aware system or link failures. It is also possible to use several Grandmasters to overcome the failure of one of the Grandmasters. This solution is less dynamic than the BMCA but unlike a network which uses the BMCA, a network using the domains with the external configuration of the ports has a very low and deterministic reconfiguration time. Indeed, in the event of a loss of synchronization information, once the loss is detected, the device can decide to set its clock to the synchronization information of another domain that is already available with gPTP. This short reconfiguration time is therefore dominated by the time to detect the failure.

Despite the fact that the fault detection function is not standardized, it is reasonable to assume that the fault detection mechanism works similarly to the one of the legacy BMCA. Thus, for a default configuration of the standard, a time-aware system has to wait for the loss of 3 consecutive `Sync` messages, i.e. 3 intervals of 125ms, to detect a failure. With the *syncLocked* mode activated, the detection of the loss takes place at the same time in all the network up to the residence time. The *syncLocked* mode forces the device to forward a `Sync` via its master ports directly upon the reception of its parent `Sync` message. We can conclude that it is possible to modify the *syncInterval* and the *syncReceiptTimeout* to change the fault detection duration in order to match our robustness constraint.

Given the lower reconfiguration time in the event of a failure and the strong design preference of the industrial world

for static configurations for safety reasons, we will therefore focus on the implementation of multiple domains using static port configuration.

With the static configuration, we have control over the state of each port in the network, and thus we can choose at set of domains that offer redundancy. A synchronization domain is mathematically represented by a spanning tree, rooted at the Grandmaster node. Ideally, to resist k failures, we should find a set of $k+1$ independent spanning trees. That is, the paths joining every pair of vertices x and y in any two spanning trees should be vertex disjoint. Finding a set with this property ensures that the loss of a link or a time-aware system will have no impact on the distribution of synchronization information, but it is difficult to observe this property on a real topology. Indeed, let us take the example of a TSN topology reminiscent of a standard automotive use case as presented in Fig. 3. In this topology, nodes 0 to 6 and 17 are switches and others are end systems. Device 4 is the Grandmaster. Two domains are defined at most since we can only find at most two independent paths between the Grandmaster and the other time-aware systems. It is possible to observe as well that end stations are not accessible by independent paths.

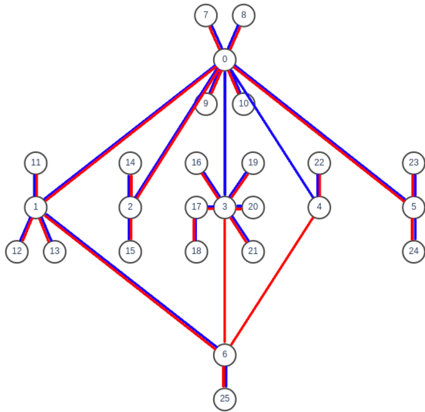


Fig. 3: A robust configuration of two domains for an automotive topology.

Ensuring the complete independence of a set of spanning trees may be intractable in real network. Thus, we are looking for the “most independent set of spanning trees”. As such, we look for instance for two spanning trees where the loss of a link or of a node prevents the distribution of synchronization information for as few nodes as possible. The configuration shown in Fig. 3 is one of the possible configurations that reduces the impact of a link or device failure compared to weaker one of Fig. 4. For example, a failure of the link between 0 and 4 has no impact on the synchronisation distribution thanks to the presence of the red domain that doesn't use this link. This is not the case for the configuration of Fig. 4. This weak configuration is also more sensitive to device failures because the distribution of `Sync` and `Follow_Up` messages is centralized by a few device such as device 0.

IV. A CONFIGURATION FOR PRECISE SYNCHRONIZATION

On top of creating a robust set of synchronisation domains, another challenge is to ensure the synchronisation service these

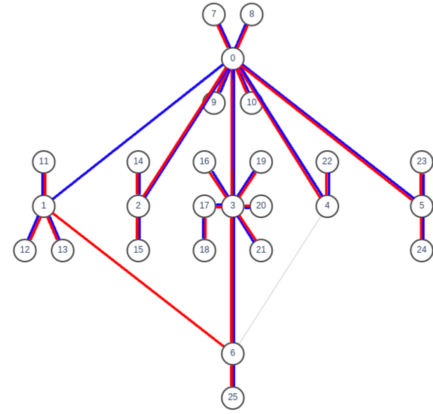


Fig. 4: A weak configuration of two domains for an automotive topology

domains offer is precise, or at least as precise as needed by the networked system. First, let's define more explicitly the term precision. Given the reference time of the Grandmaster, we can compute for any time-aware system the difference between its local time and the reference time. This is the time deviation that can be expressed or measured for any time-aware system composing the network and at any reference time. Thus, a precise device has a small time deviation from the Grandmaster. A theoretical definition of the precision of the complete networked system is given by the maximum time deviation experienced by any given time-aware system and at any time.

A. Source of imprecision

The first parameter that can impact a device's precision is the quality of its clock. Indeed, any real clock tends to drift because of tiny changes in frequency of the oscillator due to factors like aging or external temperature. If the clock drift is limited to +/- 10ppm (parts per million), the clock drifts at the maximum of +/- 1.25µs compared to an ideal clock within a *syncInterval* of 125ms.

The second parameter which can affect the precision of the synchronization is the accuracy of the time-stamping necessary for the propagation and residence time measurement mechanisms. Despite the hardware time-stamping used in devices supporting IEEE 802.1AS, which eliminates software-related inaccuracies, several phenomena impact the accuracy of timestamps such as oscillator noise, jitter caused by the PHY layer and clock granularity. In the following, we are only interested in the granularity of the clock and the PHY layer jitter. Indeed these two combined phenomena can add between 0ns and 16ns to the real timestamp, compared to the noise related to the oscillator which is less than 1ns according to P. Loschmidt et al. in [6].

B. Simulation study

In order to study the impact of inacurate timestamps on precision, simulations with an OMNeT ++ library created by H. Puttnies et al. [7] are presented next. We have added new features to this library: *i)* the rate ratio, which allows a logical

Fig. 5: The simulated daisy chain network.

syntonization with the Grandmaster, *ii)* the PHY jitter and *iii)* the clock granularity.

For our simulations, we consider a daisy chain network, as shown in Fig. 5, where 10 hops separate the Grandmaster from the most distant time-aware system. We use 5m cables between each pair of gPTP devices, enforcing a propagation time of 25ns. Clocks have a constant -10ppm drift, a PHY jitter evenly distributed between 0 and 8ns and a granularity of 8ns. The default IEEE802.1AS parameters of *syncInterval*= 125ms and the time between `Pdelay_req` of 1 second are set. Results are extracted from 30 simulations of 1000s each. The results obtained with perfect and inaccurate timestamping mechanisms are presented in the Table I.

From these results, we observe that for the perfect timestamping scenario, the worst precision is of -1250ns regardless the distance between the device and the Grandmaster. This value is the drift that a clock of -10ppm undergoes between two `Sync` messages spaced by 125ms. This time deviation can be reduced by using better clocks of lower drift or by reducing the *syncInterval* resulting in an increase in synchronization traffic. Moving to a *syncInterval* of 31.25ms and using clocks with a drift between -5ppm and 5ppm, the time deviation can be reduced to a value ranging between -162.5 and 162.5 ns.

By adding the timestamp inaccuracies in the simulations, we observe that the precision decreases with the number of hops. In fact, at each new hop, the inaccuracies in the measurement of the propagation and residence times are added to the `Follow_Up` message which is forwarded to the next device. The inaccuracies have a reduced impact on the first hop device because only the propagation measurement is needed to adjust its clock. Since we consider the Grandmaster to be perfect, due to the superior quality of its oscillator compared to other time-aware systems, the first hop device is not impacted by the Grandmaster drift or its timestamping inaccuracies. For the last jumps, the inaccuracies that accumulate are lower because the timestamping errors compensate for each other. Approaching a worst case is therefore more difficult. The error caused by the imprecision of timestamps can be reduced by using clocks with a lower granularity, by using a filter to average the propagation delay measurements or by reducing the number of jumps. Thus, the choice of the state of the different ports of the time-aware devices, and therefore of the spanning tree, impacts the precision reachable by the synchronization because of the number of hops.

Hops	Worse precision with perfect timestamp in ns	Worse precision with inaccurate timestamps in ns
1	-1250	-1259
2	-1250	-1274
3	-1250	-1281
4	-1250	-1285
5	-1250	-1288
6	-1250	-1295
7	-1250	-1301
8	-1250	-1302
9	-1250	-1307
10	-1250	-1317

TABLE I: Simulation results

V. Conclusion

This work has investigated the design of a robust and precise synchronisation service for critical on-board networks. In terms of robustness, we advocate for the use of static independent domains. We have shown that it is difficult to ensure strict independence of domains on real topologies. However, it is possible to find spanning tree sets that reduce the number of devices affected by a failure. In terms of precision, we illustrated through simulations the fact that timestamp inaccuracies cause measurement errors which propagate in the rest of the network. It is therefore reasonable to minimize the number of hops between the Grandmaster and the different time-aware system when designing a spanning tree.

Future works will investigate other sources of imprecision of the timestamp, by validating the behavior of the simulator using experimental measurements on device supporting IEEE 802.1AS and finally by proposing a method to find the sets of spanning trees that best meet the constraints of robustness and precision of on-board networks.

Acknowledgment

The authors thank all people and industrial partners involved in the EDEN project. This work is supported by the French Research Agency (ANR) and by the partners of IRT Saint Exupéry Scientific Cooperation Foundation (FCS): Airbus Operation, Airbus Defence and Space, CNES, Continental Automotive, INPT/IRIT, ISAE-SUPAERO, ONERA, Safran Electronics and Defense, Thales Alenia Space and Thales Avionics

References

[1] “IEEE Standard for Local and Metropolitan Area Networks - Timing and Synchronization for Time-Sensitive Applications in Bridged Local Area Networks,” *IEEE Std 802.1AS-2011*, pp. 1–292, 2011.

[2] “IEEE Standard for a Precision Clock Synchronization Protocol for Networked Measurement and Control Systems,” *IEEE Std 1588-2008 (Revision of IEEE Std 1588-2002)*, pp. 1–269, 2008.

[3] M. Gutiérrez, W. Steiner, R. Dobrin, and S. Punnekkat, “Synchronization quality of IEEE 802.1 AS in large-scale industrial automation networks,” in *2017 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*. IEEE, 2017, pp. 273–282.

[4] G. M. Garner, A. Gelter, and M. J. Teener, “New simulation and test results for IEEE 802.1 AS timing performance,” in *2009 International Symposium on Precision Clock Synchronization for Measurement, Control and Communication*. IEEE, 2009, pp. 1–7.

[5] “IEEE Standard for Local and Metropolitan Area Networks–Timing and Synchronization for Time-Sensitive Applications,” *IEEE Std 802.1AS-2020 (Revision of IEEE Std 802.1AS-2011)*, pp. 1–421, 2020.

[6] P. Loschmidt, R. Exel, and G. Gaderer, “Highly accurate timestamping for ethernet-based clock synchronization,” *Journal of Computer Networks and Communications*, vol. 2012, 2012.

[7] H. Puttnies, P. Danielis, E. Janchivnyambuu, and D. Timmermann, “A Simulation Model of IEEE 802.1 AS gPTP for Clock Synchronization in OMNeT++.” in *OMNeT++*, 2018, pp. 63–72.

Mutual perturbations of Fprime applications in a space context

Thomas Beck*, Frédéric Boniol†,Jérôme Ermont‡ and Luc Maillet*
*Airbus Defence and Space, Toulouse, France
Email: thomas.t.beck@airbus.com luc.maillet@airbus.com
†ONERA, Toulouse, France
Email: frederic.boniol@onera.fr
‡IRIT-ENSEEIHT, Toulouse, France
Email: jerome.ermont@toulouse-inp.fr

***Abstract*—Integration of multiple software applications in satellites avionics is part of costs reduction in the more and more competitive space industry. In this paper, we present a modelling of two space software applications. This modelling aim at identifying interferences between multiple software applications running on the same multi-core processor. Considered interferences occurred in memory (shared cache's state modification, shared bus, ...).**

I. Introduction

A. Space context

In the embedded systems industry there are not environments more hostile than space. In fact, a spacecraft is alone in the void without direct human interactions. Its only way to communicate and receive orders is through antennas. In such a context, consequences of software failures can lead to the loss of the spacecraft or even worse. For example, if the AOCS software does not work, the solar panels will not always be facing the sun and the battery will not be able to produce enough power for the whole spacecraft. In order to reduce those risks, on-board software applications have a criticality level which represents the consequences of their failures. Therefore, a failure in a high criticality level software will result in worse consequences. In the space context, criticality level for software is defined in the ECSS (European Cooperation for Space Standardization) standards as follows:

- A Catastrophic consequences: loss of human life or environment disaster.
- B Critical consequences: loss of the spacecraft and/or the mission.
- C Major consequences: major mission degradation.
- D Minor consequences: minor mission degradation.

The development of a software with a high level of criticality is more constrained and thus more expensive. Usually, to prevent that a software failure propagate to a software with a higher level of criticality, software applications are executed on different hardware platform. Most of the time, on-board software applications with the same level of criticality are not made by the same software team, and are separated on different hardware platforms as well. With this approach, software failures are contained by the hardware and software behavior does not affect execution of others software. Clearly this one software on one hardware platform strategy has a non negligible cost considering the price of an on-board computer.

B. Problem statement

This article aims at studying the cohabitation of two or more software applications on the same multi-core hardware platform. These two software applications are designed and developed according to the space context described in the previous section, thus each software application has a criticality level and is produced by a different developers team. Once they are executing on the same hardware platform, we want to assure two fundamental properties:

- **Execution interferences:** the perturbation made by one software on to others should not be greater than ϵ. This ϵ can be an execution time or a response time depending on the case.
- **Failure propagation:** a software failure should not propagate to software with a higher criticality level. The functional failure propagation, being application-specific is not in the scope of this work.

Along with these properties we assume the following hypothesis: all software should be developed and executed on Linux. It means that Linux is used as the embedded operating system of our spacecraft.

II. Space software usecases and platform

Software applications we studied exchange data between physical sensors, physical memory and the ground station. Software can be real-time, i.e with a deadline and/or a period, depending on the requirements of the sensor they communicate with. Secondly, since all software will be executed on top of a Linux operating system, a software application is represented by a Linux process but the complexity of the Linux configuration for this kind of context will not be discussed in this paper. Finally, we took two specific use-cases software to study their isolation when they are executed on the same hardware platform.

A. STR: the star tracker software

In a satellite, the star tracker is responsible for providing the satellite's attitude to the central software i.e orientation of the satellite. To do so, it is composed of several elements. The first ones are optic devices, pointed at the outer space. The role of these devices is to take pictures of space and specially stars. Once pictures are taken they will be analyzed by the second part: the star tracker software. Usually, it is executed alone on a hardware platform which can be an FPGA or a SoC or both depending on functional needs. The main goal of this piece of code is to extract the attitude data out of space pictures taken by the optic device and then send the result to the central software on the on-board computer. In this paper, we want to study the behaviour of this software when it is not alone on the hardware platform.

We choose to use the functional architecture of the star tracker software presented in [1]. The software is composed of 6 functional blocks. The first and most important block is the one which actually computes the satellite attitude. The five others blocks are support blocks and are described as follows:

- Managing the requests.
- Get the information.
- Managing the modes.
- Time management.
- Housekeeping management.

B. GPS: the GPS software

As for the STR software, the GPS software is responsible for computing the GPS sensor data into and passing them to the central software for decision making. The GPS software provides time and position to the central software. The architecture of this software is based on the one presented in the Fprime tutorial [2]. There are 8 functional blocks described as follows:

- **GPS component:** the core component which will compute data sent by the sensor.
- **Passive console text logger:** takes the text version of a log and prints it to the standard output.
- **Active logger:** serializes log entry for communication purposes.
- **Telemetry channel:** stores telemetry issued by the GPS component and sends it to the ground.
- **Command dispatcher:** receives commands and dispatches them to be executed.
- **Ground interface :** sends and receives packets from the ground.
- **Socket IP driver :** interface between the operating system socket driver and the ground interface.
- **Linux serial driver :** interface between the operating system serial driver and the GPS component.

C. Hardware platform

In the scope of this paper, all software are considered to be executed on a GR740 System on Chip. The GR740 is a radiation hardened space SoC with a 4 cores LEON4FT processor, an L2 shared cache, a memory bus, an I/O bus and peripherals. The detailed architecture is presented in [3]. Communications between cores, L2 caches, memory controller and I/O controller are made through AHB busses.

III. Fprime: a framework for space applications

Fprime is a framework used to develop avionics software designed for space applications (see [4]). Formerly developed by the NASA, it is now open source and available on GitHub. The framework provides an avionics architecture for small spacecraft (nanosat, cubesat, ...) made by students or small institutions. The main goal of Fprime is to make accessible the development of space avionics. Fprime will be used as an architecture framework for our applications. In the rest of this section we will describe the architecture of an Fprime application by explaining the framework principal concepts.

A. Components

Components are the basic modules composing an Fprime application. Each of them contains a part of the system's logic. Communication between components is made through ports and no non-port communication should be allowed. In fact, components are not aware of other components, they only see ports. There are three types of components with different functionalities and different input port types. Component types are defined as follows:

- **Passive component**: no thread and no asynchronous port invocations nor asynchronous commands. The invoking component provides the thread and the execution context while the code is inside the developer's class.
- **Active component**: has a thread and a queue. The thread dispatches port calls from the queue as on the execution context of the thread.
- **Queued component**: has no thread but does have a queue. Thus it handles asynchronous commands and port invocations; however, the user must implement at least one synchronous port invocation that unloads and handles the messages on the queue. For this and any other synchronously invocation execution context is supplied by the invoker.

B. Ports

In Fprime, ports are the only way of communication between components. Each port is signed by a specific type and can only be connected to another port signed of the same type. Components invoke ports to pass data to another component. There are four kind of ports as described in the following:

- **Output ports**: output data pass through the component output port.
- **Synchronous input**: receives synchronous invocations. All types of components can have synchronous input ports.

- **Asynchronous input**: receives asynchronous invocations via the queue of their component. Can only be defined in active and queued components.
- **Guarded input**: takes a mutex before receiving a synchronous invocation and releases it after. All types of components can have a guarded input.

C. Data constructs

A spacecraft is controlled by commands and monitored through telemetry channels and events. Fprime implements these critical data constructs for the communication in an Fprime system. Data constructs are described as follows:

- Commands are made for interaction between users and components. They are sent by users to the Fprime system and the command dispatcher passes it to the handling component.
- Events are log of activities and enhance the ability to trace the execution.
- Channels, also known as telemetry channels, represent the current reading of some portion of system state.

IV. Space software modelling

In this section, we will present a modelling of STR and GPS software constructed with the Fprime framework. Software will be described by components and ports.

A. GPS software modelling

The GPS software Fprime model is provided by the Fprime tutorial. As presented in figure 2 there are 8 components, four of them are active : CmdDispatcher, TlmChan, ActiveLogger, GPSComponent and the four others are passive components. This means that the GPS software is composed of 4 threads. In terms of ports, the GPS component has 8 ports as defined in the xml presented in figure 1:

- cmdIn: an input port used to process commands sent to this component.
- cmdRegOut: an output port used to register this component's with the command dispatcher.
- cmdResponseOut: an output port used respond to dispatched commands.
- eventOut: an output port used to send events out.
- textEventOut: an output port used to send events in a text form.
- tlmOut: an output port used to send out telemetry channels.
- serialRecv: an input port used to receive serial data buffers.
- serialBufferOut: an output port used at startup to supply buffers to the serial driver.

B. Star tracker software modelling

The STR Fprime model has almost the same architecture than the GPS one. The 6 functional blocks presented in section II are regrouped in the attitude computation component and another component named image acquisition is positioned between attitude computation and LinuxSerialDriver as shown in figure 3.

V. Problem formalisation

Our work is focused on analyzing perturbation between software applications on the same hardware platform with the properties described in the introduction of this paper. We choose to model our software using the Fprime framework. These models are described in the previous section. In this section we will discuss the missing elements in the Fprime models needed to identify interferences between different software.

First of all, even if threads are present in every active components in Fprime, there is not any possibility to manage real-time constraints within the framework. As our software applications are connected to complex sensors they sometimes need to be executed periodically and respect a deadline. Software applications are made of multiple threads which are all managed by the operating system (Linux in this case). The scheduling model of these software applications is not partitioned which means that the scheduler manages all threads all together without time-slotted partitions.

Second of all, Fprime is a high level framework, it is not possible to identify the memory allocation of each software with an Fprime description of it. The hardware part of our usecases is very important, as execution time can be modified by a memory response delayed by shared caches or busy busses. Interactions between software and hardware is made by two channels: systems calls and drivers. System calls allow software to use special features provided by the operating system. A complex operating system such as Linux proposes many different system calls and some of them might not be suitable for space avionics. In order to measure and then control interferences caused by system calls the first step is to list system calls usable in a satellite avionic. Once the list is complete an analysis of the memory impact of each system call will be made. For example, if one software uses shared pipes they can be accessed by others software. These accesses can generate interferences.

The main purpose of drivers is to abstract the handling of devices. Drivers code runs in kernel mode like the operating system and can thus access the physical memory. As they are closely related to device characteristics most of the time drivers are written by the company who developed the device. This detail poses a problem in our context, the fact that a non trusted component is running inside the kernel is a clear breach of the isolation sought in this work.

The last question raised by this modelling is about basic components of Fprime. Basic Fprime components, provided by the framework are used by both models presented in previous sections. This raises the question of whether those components are shared between two software applications or if each such component is replicated (once for each using application).

```
<port name="cmdIn" data_type="Fw::Cmd"  kind="input" role="Cmd"    max_number="1">
</port>
<port name="cmdRegOut" data_type="Fw::CmdReg" kind="output" role="CmdRegistration" max_number="1">
</port>
<port name="cmdResponseOut" data_type="Fw::CmdResponse" kind="output" role="CmdResponse" max_number="1">
```

Fig. 1. GPS ports code [2]

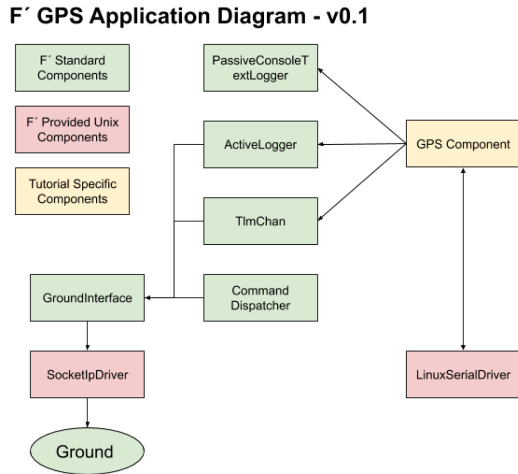


Fig. 2. Topology of a GPS software in Fprime [2]

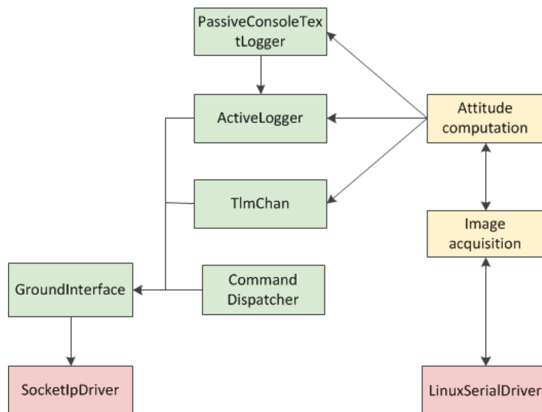


Fig. 3. Topology of a STR software in Fprime

VI. Related Work

In 2000, [5] presented core issues of IMA (integrated modular avionics) which was new at that time. In this technical report, John Rushby laid the groundwork of avionics partitioning showing that a strict isolation is a solution for resolving the IMA problematic. Many years later, solutions have been found to adapt this resolution in space avionics as shown in [6]. Hypervisors such as Xtratum and VxWorks 653 have been developed to respond to the needs of an IMA for space. Isolation proposed by this kind of hypervisor is strict, secure and with a temporal predictability. The disadvantage of those isolation properties is that it makes the whole system less modular. Scheduling is fixed and developers coding libraries are specific and not widely known and the development process is more complex.

Linux could be the perfect solution to the problems previously cited and it is used in many space mission as described in [7]. However, Linux is not a hypervisor designed to strictly isolate and to schedule real-time software applications. Linux and its PREEMPT-RT patch is able to provide real-time features. For hard real-time software applications, a co-kernel solution named Xenomai is available. The performance differences between native Linux and Xenomai are presented in [8]. Containerization is a powerful feature of Linux which helps when an isolation is required. [9] presents a way to modify the Linux scheduler to use container and real-time scheduling at the same time.

VII. Future Work

To conclude this paper we will discuss what we consider to respond to questions raised in section V. The first question is how to manage real-time in Linux with a global scheduler for all threads. The second question is what are the system authorised calls. The third question is how to handle non trusted drivers i.e drivers developed by software application developers. The fourth question is if basic components and drivers are shared by every software applications in the system.

Many threads need real-time support in a satellite avionic. The challenge is to be able to ensure that real-time constraints are satisfied when using Linux as the main operating system. To do so, Linux provides real-time features available in the kernel configurations (usually known as the PREEMPT-RT patch, which is now included by default). If those kernel features are not enough the Xenomai solution would be seriously considered. Xenomai is a co-kernel solution with a real-time kernel and a Linux kernel both running on the same hardware platform. Experiments to determine if a real-time Linux solution satisfies the on-board software real-time requirements will be an important part of the future of this work. This part of future work responds to the first question raised in section V.

One powerful feature in Linux is the containerization mostly known in software such as Docker for web development. Through namespaces and cgroups, containers can be defined to reach a certain level of space isolation. This isolation appears in different layers described in section V. First, system calls uses kernel objects that need to be isolate (semaphore, mutex, ...). Second, drivers must somehow be isolated if they are not part of the verified operating system. In the next research work, we will find if it is possible to isolate drivers inside the kernel space. Containerization could be a response to the three lasts raised questions by isolating components, system calls and drivers.

Finally, we will choose if we allow shared components and drivers in our models. This will be done by investigating the behavior of satellite software. This will be the response of our last question.

References

[1] L.-D. Stéphanie and M. David, "Str: a student developed star tracker for the esa-led esmo moon mission," Beijing, China, May - June 2010.

[2] (2021) The github repository of fprime. [Online]. Available: https://github.com/nasa/fprime

[3] *GR740 2020 Data Sheet and User's Manual*, Cobham, 2020. [Online]. Available: https://www.gaisler.com/doc/gr740/GR740-UM-DS-2-4.pdf

[4] R. Bocchino, T. Canham, G. Watney, L. Reder, and J. Levison, "F prime: an open-source framework for small-scale flight software systems," 2018.

[5] J. Rushby, "Partitioning in avionics architectures: Requirements, mechanisms, and assurance," SRI INTERNATIONAL MENLO PARK CA COMPUTER SCIENCE LAB, Tech. Rep., 2000.

[6] J. Bredereke, "A survey of time and space partitioning for space avionics," 2017.

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

[8] J. H. Brown and B. Martin, "How fast is fast enough? choosing between xenomai and linux for real-time applications," in *proc. of the 12th Real-Time Linux Workshop (RTLWS'12)*, 2010, pp. 1–17.

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

Allocation dynamique de cache mémoire dans un processeur multi-coeur avec des tâches à criticité multiple

Aléxis Génèrès[†],
[†] LAAS-CNRS
31400, Toulouse, France
Email: ageneres@laas.fr

Michaël Lauer[†], Jean-Charles Fabre[†]
[†] LAAS-CNRS
31400, Toulouse, France
Email: firstName.lastName@laas.fr

***Abstract*—Les systèmes embarqués font face à une demande toujours grandissante en puissance de calcul. L'objectif est de faire coexister des tâches à criticité multiple sur un même processeur multi-coeur. L'inconvenient de ce type de processeur est l'indéterminisme qu'il apporte. De nouvelles technologies telles que l'allocation de cache mémoire permettent de diminuer cet indéterminisme. Nous présentons, dans ce papier, une approche expérimentale pour vérifier si l'allocation de cache mémoire CAT Intel est efficace.**

***Index Terms*—multi-coeur, temps réel, criticité multiple, cache allocation**

I. Introduction

Le besoin croissant de puissance de calcul dans les systèmes embarqués conduit les industriels à utiliser massivement des processeurs multi-coeur. La multiplication des applications embarquées demande de combiner des tâches à haute criticité avec des tâches à faible criticité sur le même processeur. Malheureusement, l'utilisation de processeurs multi-coeur induit un comportement temporel non déterministe à cause de ses ressources partagées: mémoires, bus de communication... Ces facteurs peuvent provoquer des non respect d'échéances pour des tâches à haute criticité et une sous utilisation de la puissance de calcul pour les tâches à faible criticité [1].

Dans cet article, nous ciblons un seul facteur: la mémoire cache partagée entre les coeurs. Notre objectif est d'évaluer l'apport d'un mécanisme d'allocation de cache vis-à-vis du respect d'échéance temporelle. En outre, nous cherchons a évaluer si l'utilisation de ces mécanismes peut s'envisager sans trop pénaliser les tâches de fond. En effet, une allocation statique permet naturellement d'isoler les tâches temps réel et donc de limiter l'effet du partage de ressource. Cependant le cache n'est alors plus complètement disponible pour les tâches de fond même quand les tâches temps réel ne sont pas actives, ce qui de fait limite l'utilisation du processeur. Nous nous intéressons ainsi à une allocation dynamique pour éviter ce travers.

Dans ces travaux préliminaires, nous réalisons une série d'expérimentations pour évaluer l'intérêt des mécanismes de bas niveau d'allocation de cache, pour limiter l'impact, du partage de ressources dans un système à niveaux de criticité multiple mêlant des tâches à haute et faible criticité sur un processeur multi-coeur. Nous utilisons la technologie d'allocation de cache mémoire pour isoler les caches mémoire des tâches à différent niveaux de priorité.

Pour réaliser cette évaluation nous utilisons l'ordonnancement *Earliest Deadline First (EDF)* de Linux (présenté en section II). Nous nous appuyons sur les travaux de Lelli [2] qui montrent la possibilité d'utiliser 90% d'un seul coeur du CPU pour une tâche critique et garantir les exigences temps réel. De plus, nous utilisons la technologie CAT d'Intel qui est définie section II).

Pour analyser le mécanisme d'allocation du dernier niveau de cache, nous présentons dans la section II notre approche expérimentale et les outils. Ensuite dans la section III, nous décrivons notre plate-forme expérimentale et le protocole expérimental pour évaluer le mécanisme.

II. Approche expérimentale et outils

A. Approche expérimentale

Les différentes actions pour répondre à notre objectif sont:

1) de mesurer les temps de réponses de la tâche à haute criticité.
2) d'activer l'allocation de cache mémoire seulement durant le temps d'exécution de la tâche à haute criticité.
3) d'analyser le comportement qui résulte de l'allocation dynamique de cache mémoire. Nous observerons les effets de bord due à l'allocation dynamique sur les temps de réponse.

Pour la seconde action, nous mesurons les instructions par cycle (nommé IPC) de tous les coeurs et nous en faisons une moyenne générale. Nous pouvons ainsi comparer si les IPC sont plus élevés à une certaine étape, traduisant une puissance de calcul plus importante. Ainsi, les étapes peuvent être hiérarchisées selon celles qui ont le plus laissées travailler les tâches à faible criticité. Pour la troisième action, nous récupérons les temps de réponse de la tâche critique à chaque exécution. Cela nous permet de tracer les temps d'exécution de nos différentes étapes.

Si nous n'avons pas de "deadline miss" et que nous n'observons pas de changement de comportement lors de l'allocation dynamique, nous pouvons alors dire que l'allocation dynamique n'a pas d'impact négatif sur la tâche à haute criticité d'un point de vue temps de réponse. Enfin,

concernant les taches à faible criticité, nous devrions avoir une différence entre les IPCs de notre deuxième expérience et celle de la troisième. L'amélioration appréciée sera en faveur de l'utilisation de l'allocation dynamique.

B. *Outillages*

Après avoir énoncé nos objectifs, nous décrivons ici succinctement des technologies que nous utilisons. Tout d'abord, la tâche à haute criticité est ordonnancé en mode EDF de Linux (nommé *SCHED_DEADLINE*). Cet ordonnancement, sous Linux, se caractérise par trois paramètres: le temps d'exécution, l'échéance et la période (nommé en anglais respectivement *"runtime"*, *"deadline"* et *"period"*). Le temps d'exécution représente le budget accordée par le système pour réaliser la tâche avant son échéance. Ce budget est attribuée à chaque nouvelle période. Si ce budget est entièrement consommé, la tâche est dite suspendue. Du coup, elle est considérée comme ayant violé son échéance. Concernant les tâches à faible criticité nous laissons l'ordonnancement par défaut qui est: ordonnancement à temps partagés.

Pour l'allocation de cache mémoire, nous utilisons la technologie CAT d'*Intel*[3]. La technologie CAT (Cache Allocation Technology) permet d'autoriser ou d'interdire l'écriture pour un groupe de processus ou de coeurs dans un sous espace du dernier niveau de caches du CPU. Ce sous espace est définie par le nombre de *ways* du dernier niveau de cache. Les caches mémoires des processeurs sont découpées en un nombre finis de *ways* qui sont des tables de mémoire. Par exemple, pour un dernier niveau de cache possédant 8 *ways*, nous pourrons alors allouer de un à huit *ways* (zéro étant impossible).

Enfin pour obtenir les données (d'instructions par cycle et les temps de réponse) nous utilisons BCC (BPF Compiler Collection)[4]. BCC nous permet de programmer nos outils d'observation et de mesure. C'est un ensemble d'outils qui permettent de programmer et d'utiliser les sondes BPF (Berkeley Packet Filter) afin de collecter les données nécessaires à notre expérimentation. BCC permet une instrumentation du noyau à grain fin et n'engendre qu'une faible surcharge de calcul.

III. Protocole expérimental

Pour répondre à notre problématique nous réalisons une expérience en trois étapes. Chaque étape sera constituée d'une exécution de la tâche à haute criticité seule puis, dans un second temps, une exécution de la tâche à haute criticité et les tâches à faible criticité en parallèle pour induire des interférences. Les trois étapes sont :

- Sans allocation de cache mémoire.
- Avec une allocation statique de cache mémoire.
- Avec une allocation dynamique de cache mémoire.

Avant de détailler pas à pas le protocole expérimental, nous présentons le matériel et les outils que nous utilisons.

Pour notre expérience, nous disposons d'une machine intégrant deux processeurs Intel Xeon Bronze 3204. Chacun de ces processeurs possède 6 coeurs. Chaque processeur possède un dernier niveau de cache de 8.25MB en 11 ways associative. Les caches mémoires L2 sont d'une taille de 1 MB chacun et ne sont pas partagés entre les coeurs.
Avec les informations sur les derniers niveaux de cache, nous pouvons détailler l'utilisation de CAT dans ce cas. Nous avons deux fois 11 *ways* d'allocation possible. Chaque *way* représente 0.75 MB d'espace mémoire. Nous utiliserons donc ce mécanisme pour isoler notre tâche à haute criticité en lui allouant un nombre de *ways* suffisant. Et en supprimant l'autorisation d'écrire dans les *ways* allouées pour les tâches à faible criticité.

Concernant le système d'exploitation, nous travaillons sur un noyau 5.12.5 pour utiliser les versions les plus récentes de BCC. Nous utilisons BCC pour sauvegarder la date de lancement et de fin de la tâche haute criticité. Aussi, nous devons enregistrer toute violation d'échéance de la tâche à haute criticité. Enfin, pour les tâches à faible criticité, nous mesurons le nombre d'Instructions Par Cycle (IPC) de chaque coeur.

A. *Protocole détaillé*

Chacune des étapes (sans allocation, avec une allocation fixe et enfin avec une allocation dynamique) est constituée de deux sous étapes; une avec la tâche à haute criticité seule, puis une avec la tâche à haute criticité et les tâches à faible criticité en parallèle.
Comme nous utilisons l'ordonnancement EDF Linux pour la tâche à haute criticité, nous devons fixer les trois paramètres de cet ordonnancement: le temps d'exécution, l'échéance et la période.

Pour notre expérience nous fixons un temps d'exécution de 150ms et une période, égale à l'échéance, de 200 ms. Nos tâches de haute et faible criticité utilisent un volume données de 3 MB.Pour observer l'effet de la mémoire partagée, le volume des données doit être plus grands que la taille du cache L2, qui est de 1 MB.

1) Étape sans allocation de cache mémoire: Cette étape est composée de deux sous-étapes. La première, est de lancer la tâche à haute criticité seule avec les outils d'observations. Cette sous-étape vise à collecter les temps de réponse de référence pour la tâche à haute criticité.
Lors de la seconde sous-étape, nous relançons la tâche de haute criticité mais en exécutant cette fois les tâches de stress, de faible criticité, en parallèle. Nous lançons 12 tâches de stress, une par coeur. Ainsi, nous pouvons observer l'effet des voisins bruyants (noisy neighbours) sur notre système à savoir des temps de réponse plus long. Ce terme voisins bruyants correspond aux interférences dues à la mémoire cache partagée utilisée par d'autres applications. Durant cette sous-étape, nous recueillons également l'IPC de référence.

2) Étape avec allocation de cache mémoire fixe: L'allocation de cache mémoire fixe reste effective jusqu'à la fin de cette étape. Pour la mettre en place, nous allouons cinq *ways* successifs sur les 22 disponibles (chaque processeur possédant 11 *ways*) à la tâche à haute criticité, via le

mécanisme CAT.
Cette réservation sera exclusive à la tache de haute criticité. Nous réservons une taille mémoire de 3.75 MB (0.75 par *way*). En effet, elle doit être suffisamment grande pour que les 3 MB de données de la tâche à haute criticité soit protégées.

Comme pour l'étape précédente, nous faisons deux sous étapes dans cette configuration. La première, sans les tâches à faible criticité, nous permet d'observer l'impact de la réservation de cache sur les temps de réponse de la tâche à haute criticité. L'impact est censé être minime, voire nul.
Ensuite, nous chargeons le système avec les 12 taches de faible criticité. Nous ne devrions pas avoir de modification des temps de réponse car l'allocation de mémoire protège notre tâche à haute criticité. Cependant, nous supposons que le retrait des cinqs *ways* aux tâches de faibles criticité aura un impact négatif sur les IPC mesurés.

Pour observer le résultat nous comparerons:

- les temps de réponse moyen et maximum.
- les IPC moyens de tous les coeurs.

3) Étape avec allocation de cache mémoire dynamique: Nous conservons les cinq *ways* d'allocation précédemment attribués mais nous désactivons l'allocation une fois la tâche critique terminée. Nous réactivons l'allocation de cache mémoire à la nouvelle période de la tâche à haute criticité.

Pour répondre à l'action 2) nous utilisons l'expérimentation sans tâches de stress. Pour analyser l'impact de l'allocation dynamique nous tracerons, dans un graphe, les évolutions temporelles des temps de réponses avec et sans allocation dynamique.

Enfin, pour répondre à l'action 3) nous ajoutons nos tâches de stress pour comparer les IPC calculés avec les IPC de l'étape 2). La différence de ces deux IPC moyen conclue sur la rentabilité du mécanisme d'un point de vue des tâches à faible criticité.

IV. Travaux connexes

Dans cette partie nous allons voir des solutions pour faire cohabiter des tâches à haute et à faible criticité sur un même processeur multi-coeur et en voir leurs limites.
Une première solution illustrée par les travaux de Suzuki et al [5] propose de réserver pour chaque coeur un espace du cache mémoire. Ensuite en affectant un coeur aux tâches à haute criticité et les autres aux tâches à faible criticité, il réussi à limiter l'impact de cette ressource partagée. Cependant cette solution est assez pessimiste car il est impératif d'affecter un coeur exclusivement aux tâches à haute criticité et, il y a un risque que l'espace de cache mémoire réservé ne soit pas utilisé. Un autre type de solution, que nous appellerons solution "tout ou rien" consiste à exécuter les tâches à hautes criticité en désactivant toutes les autres tâches à faibles criticités. C'est le cas du travail de Kritikakou et al. [6] qui montre comment garantir les exigences temps réel en utilisant une approche en deux étapes. Dans un premier temps, les tâches à haute criticité sont exécutées en parallèle des tâches à faible criticité. Ensuite, en se basant sur le RWCET (remaining worst case execution time), les tâches à faible criticité sont désactivées. Dans nos travaux, nous souhaitons éviter de désactiver les tâches à faible criticité.
D'autre travaux de Kritikakou et al. [7] et de Girbal et al.[8] utilisent cette méthode.
Enfin, une solution proposé par Xu et al [9] consiste à réserver en groupe de priorité des parties du cache mémoire, puis à les allouer virtuellement. Pour éviter les interférences, les tâches à faible criticité sont isolé par la technologie CAT en leur allouant un sous espace de la mémoire cache. Le point faible de cette technique est que les tâches à faible criticité ne peuvent pas utiliser 100% du cache mémoire disponible due à la pré-réservation.

V. Conclusion

Pour conclure, nous analysons le mécanisme d'allocation dynamique du dernier niveau de cache avec un poste expérimental équipé de deux processeurs intel, sur Linux. Cette démarche nous permet de statuer sur la viabilité de ce type de mécanisme pour une tâche temps réel d'une durée d'exécution de l'ordre de 100ms. De plus notre approche cherche à limiter l'impact sur les tâches à faible criticité. Actuellement, nous développons les outils d'allocations dynamique pour la plate-forme expérimentale. Après obtention des résultats expérimentaux selon l'approche qui a été proposée dans cet article, une prochaine étape sera de rajouter des autre tâches à haute criticité. Puis, la suite consistera à utiliser plus d'un coeur pour les tâches à haute criticité.

References

[1] C. Cullmann and al., "Predictability considerations in thedesign of multi-core embedded systems," 2010. [Online]. Available: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.172.4533&rep=rep1&type=pdf

[2] A. L. Lelli Juri, Scordino Claudio and F. Dario, "Deadline scheduling in the linux kernel," *Software practice and experience*, vol. 46, 2015. [Online]. Available: https://onlinelibrary.wiley.com/doi/epdf/10.1002/spe.2335

[3] K. T. Nguyen, "Cat cache allocation technology," 2016. [Online]. Available: https://software.intel.com/content/www/us/en/develop/articles/introduction-to-cache-allocation-technology.html

[4] "Bpf compiler collection (bcc)," 2021. [Online]. Available: https://github.com/iovisor/bcc

[5] N. Suzuki, H. Kim, D. d. Niz, B. Andersson, L. Wrage, M. Klein, and R. Rajkumar, "Coordinated bank and cache coloring for temporal protection of memory accesses," in *2013 IEEE 16th International Conference on Computational Science and Engineering*, 2013, pp. 685–692.

[6] A. Kritikakou, T. Marty, and M. Roy, "DYNASCORE: DYNAmic Software COntroller to Increase REsource Utilization in Mixed-Critical Systems," *ACM Tran. on Design Automation of Electronic Systems*, vol. 23, 2017. [Online]. Available: http://dl.acm.org/citation.cfm?doid=3149546.3110222

[7] A. Kritikakou, C. Rochange, M. Faugère, C. Pagetti, M. Roy, S. Girbal, and D. G. Pérez, "Distributed run-time wcet controller for concurrent critical tasks in mixed-critical systems," in *Proceedings of the 22nd International Conference on Real-Time Networks and Systems*, ser. RTNS '14. New York, NY, USA: Association for Computing Machinery, 2014, p. 139–148. [Online]. Available: https://doi.org/10.1145/2659787.2659799

[8] S. Girbal, X. Jean, J. Le Rhun, D. G. Pérez, and M. Gatti, "Deterministic platform software for hard real-time systems using multi-core cots," in *2015 IEEE/AIAA 34th Digital Avionics Systems Conference (DASC)*, 2015, pp. 8D4–1–8D4–15.

[9] M. Xu, L. Thi, X. Phan, H.-Y. Choi, and I. Lee, "vcat: Dynamic cache management using cat virtualization," in *2017 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2017, pp. 211–222.

Dimensionnement des buffers en nombre de trames dans l’Ethernet commuté

Richard Garreau, Frédéric Ridouard, Henri Bauer, Pascal Richard
LIAS- Université de Poitiers et ISAE-ENSMA
Poitiers, France
Email: {richard.garreau, pascal.richard}@univ-poitiers.fr, {frederic.ridouard, henri.bauer}@ensma.fr

***Résumé*—Les systèmes temps réel critiques sont de plus en plus distribués et nécessitent une preuve de déterminisme de plus en plus complexe. Le dimensionnement des files d’attente dans un réseau temps réel critique fait partie de cette preuve. En règle générale, ce dimensionnement est statique pour ce type de systèmes. Néanmoins, il existe peu de solutions apportant une réponse à ce problème. Dans ce papier, nous traiterons de la problématique du dimensionnement statique des files d’attente dans les switchs Ethernet et de la méthode que nous avons développée spécifiquement pour répondre à ce besoin : *Buffer Dimensioning per Frame* (BDF).**

I. Introduction

Les systèmes embarqués temps réel sont composés de plus en plus de fonctions avec pour conséquence une augmentation des échanges des données. Une solution à cette évolution des systèmes temps réel critique est une architecture distribuée basée sur un réseau hautes performances Ethernet. Des protocoles basés sur Ethernet sont développés pour être temps réel (AFDX, AVB, TSN, …). Le déterminisme de ces réseaux doit être démontré et en particulier le non-débordement des files d’attente (*buffers*).

Deux approches permettent de dimensionner les files d’attente : par bits ou par nombre de trames. Lorsque la mémoire est définie en nombre de bits, l’allocation est dite dynamique et le nombre de bits alloués se fait à la demande. En allocation statique, la mémoire est découpée en nombre de slots (ou de trames pour une file d’attente) de taille identique et fixe. À chaque demande, un slot est alloué. Les slots sont de taille suffisamment grande pour permettre l’allocation d’une trame de taille maximale. Dans les systèmes critiques, l’allocation est généralement statique. Cependant, déterminer la taille mémoire en nombre de trames est difficile.

Une façon simple de déterminer cette occupation en trames est de diviser la pire occupation en bits par la taille de trame minimale. Cette approche est pessimiste, c’est pourquoi nous proposons une nouvelle méthode pour dimensionner les files d’attente en nombre en trames: *Buffer Dimensioning per frame* (BDF). Cette méthode prend en entrée : une topologie réseau, un contrat de trafic à l’entrée du réseau pour chaque flux, ainsi qu’une méthode de calcul de délais de bout en bout.

Dans le paragraphe II, nous aborderons l’état de l’art du dimensionnement des buffers. Ensuite nous décrirons nos motivations pour la méthode BDF dans le paragraphe III. Nous décrirons le modèle réseau paragraphe IV. La méthode BDF sera détaillée dans le paragraphe V ainsi que son application sur un petit exemple paragraphe VI.

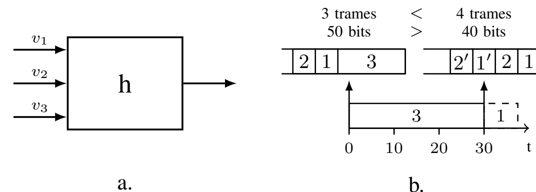


Figure 1. Différences entre occupation d’un buffer en bits et en nombre de trames

II. État de l’art

Dans la littérature, il existe peu de travaux concernant le dimensionnement des buffers dans les réseaux temps réel. Les travaux existants se basent généralement sur les méthodes d’analyse de délais de bout en bout pire cas. En effet, le problème d’analyse du délai et du dimensionnement des files d’attente sont connexes. L’arriéré de travail (en bits) présent dans la file d’attente correspond au délai d’attente multiplié par la vitesse de transmission en sortie de la file d’attente.

D’autres approches ont été développées telles qu’une approche reposant sur la méthode TA [1]. Appliquée sur un réseau AFDX, elle permet d’obtenir l’arriéré de travail (quantité maximale de données) [2] dans les files d’attente. Le nombre de trames est ensuite obtenu en supposant que toutes les trames font la même taille.

III. Motivations

Déterminer la taille d’une file d’attente en termes de trames est un problème différent du calcul en termes de bits. En effet, les instants auxquels on obtient l’occupation pire cas ne coïncident pas nécessairement.

La Figure 1 représente un cas simple exhibant trois flux traversant un nœud h : v_1, v_2, v_3 ainsi que l’état des files d’attente au cours du temps. On suppose, sans perte de généralité, que la file d’attente de h est vidée à un débit constant de 1 bit par µs. Les tailles des trames générées par v_1 et v_2 sont de 10 bits tandis que celles de v_3 ont une taille de 30 bits. À l’instant 0, une trame de chaque flux sont dans la file d’attente pour une occupation de 50 bits. La trame 3 commence sa transmission dès $t = 0$ par h. La transmission de la trame 3 (du flux v_3) dans h débute en $t = 0$ et se termine en $t = 30$. Durant ce temps deux nouvelles trames de v_1 et v_2 (notées $1'$ et $2'$) sont arrivées dans la file d’attente. Nous avons à ce moment-là une occupation de 4 trames pour seulement 40 bits.

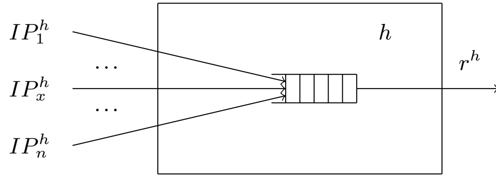


Figure 2. Représentation d'un nœud h dans le modèle réseau

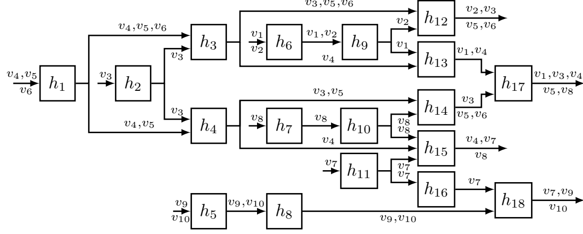


Figure 3. Construction d'un réseau en utilisant le modèle décrit

Nous présentons dans la suite une méthode développée spécifiquement pour le dimensionnement des buffers en nombre de trames.

IV. Modèle réseau

Le réseau est composé d'un ensemble de nœuds S traversés par un ensemble de flux Γ. Γ^h correspond à l'ensemble des flux traversant le nœud h ($h \in S$).

Un nœud h est composé de n liens entrants notés IP_x^h (avec $n \in \mathbb{N}^*$ et $1 \leq x \leq n$), une file d'attente gérée avec la politique de service FIFO et un lien de sortie de débit r^h. L'ensemble de flux venant du lien entrant IP_x^h est noté Γ_x^h ($1 \leq x \leq n$) (cf. Figure 2).

Les flux $v_i \in \Gamma$ sont sporadiques, émis depuis un nœud source ($first_i$) vers une destination ($last_i$) et parcourent un chemin ordonné de nœuds statiques P_i. L'ensemble des flux générés par un nœud h est noté Γ_0^h. Nous avons donc : $\Gamma^h = \bigcup_{x=0}^{n} \Gamma_x^h$ avec $n \in \mathbb{N}$

Chaque flux $v_i \in \Gamma$ est contraint par un contrat de trafic à l'entrée du réseau composé de :

- La taille maximale d'une trame : $Fmax_i$ où v_i est le flux étudié. Nous en déduisons le temps de transmission maximum d'une trame $v_i \in \Gamma$ dans un nœud h: $C_i^h = \frac{Fmax_i}{r^h}$
- Le temps minimum d'attente entre l'émission de deux trames dans le nœud source : T_i.

Nous notons $Smin_i^h$ (resp. $Smax_i^h$) le temps minimum (resp. maximum) pour une trame d'un flux v_i entre sa génération sur $first_i$ et son arrivée sur h. Nous en déduisons la gigue réseau $J_i^h = Smax_i^h - Smin_i^h$.

Un exemple de réseau compatible avec ce modèle est donné Figure 3. Il est composé de 10 flux et de 18 nœuds.

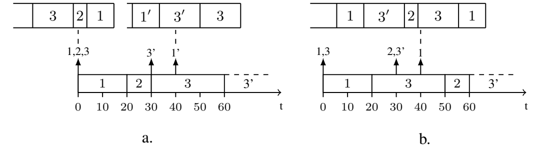


Figure 4. Différents scénarios d'arrivée dans une file d'attente FIFO

V. La méthode BDF

Nous allons dans cette section présenter le fonctionnement de la méthode BDF, pour le calcul du nombre de trames pire cas dans une file d'attente. Nous rappelons que nous avons besoin en entrée d'une topologie réseau Γ, d'un ensemble de flux régulés par un contrat de trafic et d'une méthode quelconque de calcul réseau permettant d'obtenir la gigue J_i^h.

A. Principe

La borne supérieure sur le dimensionnement en nombre de trames de la file d'attente du nœud h est notée NbF^h. Le nombre de trames stockées simultanément dans un nœud peut varier en fonction du scénario d'arrivée des trames. Il n'y a, à notre connaissance aucun travail sur la construction d'un scénario menant à maximiser le nombre de trames dans une file d'attente. BDF doit utiliser un scénario permettant d'obtenir l'occupation maximale des *buffers*. Pour le déterminer, le nombre de trames entrantes est maximisé et le nombre de trames sortantes est lui minimisé. Il est possible d'améliorer les résultats de ce scénario en prenant en compte la sérialisation des trames qui partagent le même lien.

Nous illustrons ce problème sur la Figure 4. Nous reprenons ici, la même configuration que dans la Figure 1 avec h utilisant la politique de service FIFO. Les flèches représentent l'arrivée de trames dans la file d'attente. Dans le scénario de 4.a, nous obtenons un dimensionnement pire cas NbF^h de 3 trames aux instants $t = 0$µs et $t = 40$µs. Si, nous déplaçons l'arrivée de la trame 2 de v_2 à 30µs (Figure 4.b) nous obtenons un dimensionnement pire cas NbF^h de 4 trames à $t = 40$µs.

B. Maximisation du nombre de trames entrantes pire cas

Le nombre maximum pire cas de trames entrantes peut être borné par la *Request Bound Function* (RBF) [3]. La fonction $rbf_i^h(t)$ défini pour tout intervalle $[0, t]$, le temps total de transmission de trames générées par 1 flux v_i sur le nœud h.

$$rbf_i^h(t) = \left(1 + \left\lfloor \frac{t+J_i^h}{T_i} \right\rfloor\right) C_i^h \text{ avec } J_i^h = Smax_i^h - Smin_i^h$$

La détermination du nombre de trames pire cas arrivant dans h sur l'intervalle $[0, t]$ se fait en particulier avec le terme : $\left(1 + \left\lfloor \frac{t+J_i^h}{T_i} \right\rfloor\right)$.

C. Minimisation du nombre de trames sortantes pire cas

La minimisation du nombre de trames sortantes est un problème complexe. Toutefois, nous utiliserons l'algorithme *Largest Processing Time First* (LPT) [4]. LPT est l'algorithme non préemptif qui à chaque instant choisit de transmettre une trame de taille max. L'utilisation de cet algorithme LPT

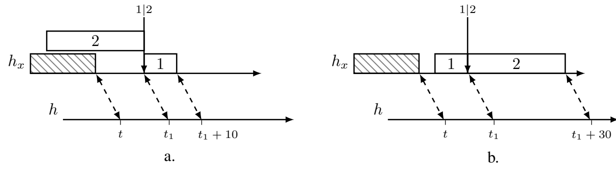


Figure 5. Comparaison de scénario d'arrivée de trames

permet de minimiser la borne minimale du nombre de trames sortantes.

D. Effet de la sérialisation

Comme décrit dans le modèle réseau, les trames arrivent par un ensemble de liens. Les trames arrivant par un lien n'arrivent pas simultanément dans le nœud, mais l'une après l'autre. C'est l'effet de sérialisation [5]. La prise en compte de cet effet permet de réduire le pessimisme dans le décompte des trames entrantes dans un nœud h (cf. paragraphe V-B). L'arrivée de deux trames consécutives dans un lien est séparée au moins par un délai de transmission de trame. Étant donné que les trames transmises dans cet intervalle libèrent des slots dans la file d'attente, on obtient un décompte moins pessimiste du nombre de trames.

L'ordre dans lequel les trames entrent peut avoir une conséquence sur l'occupation pire cas des files d'attente. Il faut donc considérer la stratégie maximisant le nombre de trames à l'entrée d'un nœud. La politique *Shortest Processing Time First* (SPT), qui sert en priorité les trames les plus petites va nous servir de base. L'usage de cette politique seule n'est pas suffisant pour déterminer un scénario pire cas. Pour décrire le scénario pire cas d'arrivée de trames, nous devons introduire une nouvelle notion : la période d'activité.

Définition 1: La période d'activité est la période entre deux instants oisifs consécutifs. Un instant oisif est un instant où toutes les trames de la file d'attente ont été servies.

La politique SPT seule ne correspond pas au scénario pire cas lorsqu'il y a présence d'un instant oisif. En effet, cet instant oisif aurait pu ne pas exister si une trame plus grande avait été choisie à la place de la plus petite.

La Figure 5, décrit deux possibilités d'ordonnancement pour deux trames de flux v_1 (10 bits) et v_2 (30 bits). La flèche verticale avec au sommet "1|2" indique le temps minimal pour que le nœud h_x ait fini de transmettre l'une des deux trames. Sur 5.b SPT est appliquée et la trame 1 est choisie pour être émise étant donné qu'entre 1 et 2, elle est la plus petite. En procédant de la sorte, un instant oisif est généré et le temps requis pour transmettre 1 et 2 correspond à $t_1 + 30$. Tandis que sur 5.a la trame la plus grande (2) est transmise la première. Cela empêche la génération du temps oisif étant donné que le temps de cet instant oisif est plus petit que le temps de transmission de la trame de taille maximum. Les trames sont transmises à h au temps $t_1 + 10$. En suivant ce scénario, nous avons fait passer deux trames au lieu d'une seule et donc maximisé le nombre de trames à transmettre. Notons néanmoins que cet intervalle n'est pas suffisant pour transmettre entièrement la trame de v_2. Malgré que ce cas soit impossible dans la réalité, il sera plus pessimiste que tous les scénarios qui peuvent réellement exister.

Nous en avons déduit un scénario qui maximise le nombre de trames arrivant dans un nœud h en deux points.

- L'arrivée des trames suit la politique de service SPT tant que de nouvelles trames sont disponibles.
- À chaque temps oisif sur un lien d'entrée d'un nœud, on insert une trame de taille maximale avant de repasser à SPT.

E. Condition suffisante pour prouver que l'application de BDF est finie

La valeur de NbF^h peut évoluer à chaque fois qu'une trame est émise. S'il y a présence d'un instant oisif aucune des trames arrivant par la suite ne peut influer l'arriéré de travail déjà calculé pour la période d'activité précédente. Par conséquent, si nous trouvons un instant oisif, nous savons que l'exécution de la méthode pourra s'arrêter.

Nous pouvons savoir qu'un instant oisif existe pour un nœud h si la condition suivante est respectée :

$$U^h = \sum_{v_i \in \Gamma^h} \frac{C_i^h}{T_i} < 1$$

Si U^h est plus petit que 1, cela signifie que la quantité de données transitant par h est inférieure à la quantité que ce dernier peut transmettre et donc en dépit des rafales de trames que le nœud peut subir, il finira par connaître un instant oisif.

F. Algorithme BDF

Afin de décrire l'algorithme BDF, nous devons introduire de nouvelles notations :

- $W^h(t)$: La fonction comptant le nombre de trames ayant été admises dans le nœud h sur l'intervalle $[0, t]$ en suivant le scénario considérant la sérialisation.
- $W_x^h(t)$: La fonction comptant le nombre de trames ayant été admises dans le nœud h par un lien x sur l'intervalle $[0, t]$.
- $W_{out}^h(t)$: La fonction décrivant le nombre de trames servies par le nœud h sur l'intervalle $[0, t]$ en suivant la politique LPT.

La fonction $W^h(t)$ est définie par :

$$W^h(t) = \sum_{v_i \in \Gamma_0^h} \left(1 + \left\lfloor \frac{t + J_i^h}{T_i} \right\rfloor\right) + \sum_{x=1}^{n} W_x^h(t)$$

La somme à gauche correspond aux trames générées localement par le nœud h, tandis que celle de droite correspond à la somme des trames qui arrivent sur les liens entrants IP_x^h.

Nous détaillons le principe général du calcul avec l'algorithme 1. Le cœur de la méthode réside dans la boucle *tant que*. $W^h(t) - W_{out}^h(t)$ correspond à la différence entre le nombre de trames admises et émises sur l'intervalle $[0, t]$. NbF^h est défini comme la valeur maximale de cette différence pour tout t. Le calcul de t permet de discrétiser le temps pour limiter les calculs aux instants d'arrivées de nouvelles trames selon la rbf (cf. V-B).

	v_1	v_2	v_3	v_4	v_5	v_6	v_7	v_8	v_9	v_{10}
$Fmax_i$ (bits)	1000	2000	1000	1000	2000	2200	6400	2200	1000	1000
T_i (s)	60	120	100	80	60	60	120	100	80	80

Table I. CARACTÉRISTIQUES DES FLUX CIRCULANTS DANS LA TOPOLOGIE RÉSEAU DÉCRITS DANS LA FIGURE 3

Entrées : Un réseau défini par S et Γ où la condition sur U^h est respectée
Sorties : NbF^h pour tous les nœuds $h \in S$

1 **début**
2 **pour chaque** *nœud* $h \in S$ **faire**
3 **pour chaque** *flux* $v_i \in \Gamma^h$ **faire**
4 $J_i^h \leftarrow Smax_i^h - Smin_i^h$
5 **fin**
6 $NbF^h \leftarrow 0$
7 $t \leftarrow 0$
8 **faire**
9 $NbF^h \leftarrow max(NbF^h, W^h(t) - W_{out}^h(t))$
10 $t \leftarrow$ $\min\left\{t_i \leftarrow -J_i^h + nT_i | n \geq 0, t_i > t, v_i \in \Gamma_x^h\right\}$
11 **tant que** $W^h(t) \geq W_{out}^h(t)$;
12 **fin**
13 **fin**

Algorithme 1 : Algorithme *Buffer Dimensioning per Frame* (BDF)

	Méthode Naive		BDF
	Arriéré (bits)	Nombre de trames (trames)	NbF^h (trames)
h_1	5200	6	3
h_2	1000	1	1
h_3	5400	6	5
h_4	3000	3	3
h_5	2000	2	2
h_6	3000	3	2
h_7	2200	1	1
h_8	1000	1	1
h_9	2000	2	2
h_{10}	2200	1	1
h_{11}	6400	1	1
h_{12}	10200	11	8
h_{13}	2000	2	2
h_{14}	4200	5	4
h_{15}	9600	10	4
h_{16}	6400	1	1
h_{17}	9600	10	9
h_{18}	7400	8	3

Table II. COMPARAISON ENTRE BDF ET L'APPROCHE NAÏVE SUR LA CONFIGURATION DONNÉE EN EXEMPLE

VI. ÉVALUATION EXPÉRIMENTALE

Dans cette section, nous détaillerions le calcul de la méthode BDF sur un exemple. Nous utiliserons la topologie détaillée dans la Figure 3 avec les flux décrits dans la Table I avec la méthode de calcul réseau *Forward end-to-end delay Analysis* (FA)[3]. Le détail du calcul de délai ne peut pas être développé dans ce papier pour des raisons de place. Le calcul de FA pourrait être remplacé par toute autre méthode pire cas. Les résultats de l'application de la méthode sur ce réseau est montré sur la Table II.

Nous allons nous concentrer sur le dimensionnement de la file d'attente du nœud h_3 (Figure 3). Ce nœud est parcouru par les flux v_3, v_4, v_5, v_6 qui arrivent par deux liens entrants. Aucune trame n'est générée par le nœud, donc : $\Gamma_0^h = \emptyset$ Nous obtenons les gigues suivantes avec la méthode FA :
$J_3^{h_3} = 0, J_4^{h_3} = 42, J_5^{h_3} = 32, J_6^{h_3} = 30$.
Nous avons $W^h(t) = 0 + SPT_1^{h_3}(t) + SPT_2^{h_3}(t)$, avec dans le lien 1 : les flux $v3, v5, v6$ et dans le lien 2 : le flux v_2. À l'instant 0, nous avons par application de l'algorithme BDF, nous avons 4 trames présentes dans la file d'attente. Par l'application de la méthode pour tout t, nous obtenons le maximum $NbF^{h_3} = 5$ à l'instant 90. La méthode naïve donne 6 trames.

Pour les résultats globaux de la table II. Nous pouvons remarquer que les résultats de BDF avec la méthode FA ne sont jamais plus pessimistes que les résultats obtenus avec la méthode naïve. Nous remarquons une différence importante entre les deux approches pour le nœud $h15$. Cela peut s'expliquer par la présence de trames de taille très différentes pour un arriéré grand. Le résultat est pessimiste pour la méthode naïve, car la différence entre les plus grandes trames et les plus petites sont élevés et la méthode naïve essaiera de maximiser le nombre de trames de taille minimale dans l'arriéré.

VII. CONCLUSION

Avec la méthode *Buffer Dimensioning per Frame*, nous sommes donc capables de déterminer le dimensionnement des files d'attente en nombre de trames. La méthode peut être utilisée avec n'importe quelle méthode de calcul de délais réseau. Elle repose sur l'observation des entrées et des sorties de trames dans les files d'attente selon un scénario pire-cas, quelle que soit la politique de service choisie du moment qu'elle est conservative. De plus, la méthode est rendue plus efficace avec la gestion de la sérialisation. Elle est plus efficace que les méthodes naïves de dimensionnement. Ces travaux pourront être réutilisés à des fins de certification dans un contexte de systèmes embarqués temps réel critique.

REFERENCES

[1] H. Bauer, J.-L. Scharbarg, and C. Fraboul, "Improving the worst-case delay analysis of an afdx network using an optimized trajectory approach," *IEEE Transactions on Industrial informatics*, vol. 6, no. 4, pp. 521–533, 2010.

[2] R. Coelho, G. Fohler, and J.-L. Scharbarg, "Dimensioning buffers for afdx networks with multiple priorities virtual links," in *2015 IEEE/AIAA 34th Digital Avionics Systems Conference (DASC)*. IEEE, 2015, pp. 10A5–1.

[3] B. Nassima, R. Frédéric, B. Henri, and R. Pascal, "Forward end-to-end delay for afdx networks," *IEEE Transactions on Industrial Informatics*, vol. 2018, pp. 858–865, 2018.

[4] R. L. Graham, "Bounds on multiprocessing timing anomalies," *SIAM journal on Applied Mathematics*, vol. 17, no. 2, pp. 416–429, 1969.

[5] A. Bouillard, N. Farhi, and B. Gaujal, "Packetization and Aggregate Scheduling," INRIA, Research Report RR-7685, Jul. 2011. [Online]. Available: https://hal.inria.fr/inria-00608852

ETR 2021