Real-Time Polling Task: Design and Analysis

Benoit Varillon, David Doose

Real-Time Polling Task: Design and Analysis

Benoit Varillon
Jean-Baptiste Chaudron
ISAE-SUPAERO
Université de Toulouse, France
Email: benoit.varillon@isae-supaero.fr
jean-baptiste.chaudron@isae-supaero.fr

David Doose
Charles Lesire
ONERA/DTIS
Université de Toulouse, France
Email: david.doose@onera.fr
charles.lesire@onera.fr

***Abstract*—Usually, robotic systems must be designed as reactive systems because they must be able to react and adapt to their environment and also communicate with some other systems. However, the reactive behavior, because of its non-deterministic nature, is difficult to analyze and prevents designers to perform a proper real-time analysis which is usually needed for critical robotic systems. In this paper, we propose a deterministic task model to handle the reactive behavior as well as the necessary tools to analyze it and verify the respect of real-time constraints. An implementation of this model, which is used in a ROS2 patch, is also presented.**

***Index Terms*—Timing predictability, Formal methods, Modeling, Real-time Analysis, Validation**

I. Introduction

The aim of autonomous systems is to evolve in an uncontrolled environment. To achieve this goal, the system needs to be able to interact with its environment, because it is continuously evolving and can't be controlled, it is important that actions are still relevant when they are executed and not due to out-of-date information. Not respecting this constraint can lead to unwanted situations with damage to the system or even to people if the system evolves alongside human. Such systems that can cause hazardous situations if they behave poorly are called critical systems.

When designing such system, it is important to be able to model it using known abstractions in order to validate safety properties. For real-time systems the model needs to have a deterministic behavior, without determinism prediction of the potential executions will not be possible and real-time analysis will be difficult with classical methods. In robotic applications, the system is often subject to external requests that are not predictable, thus there is a need for a model that can represent this type of interaction where the system needs to react to unsteady arrival of messages. The best model, in terms of latency, is a reactive model where the task is activated as soon as the requests arrive. However, a fully reactive model is not deterministic.

Scheduling theory proposes multiple models and methods to analyze the timing behavior of real-time systems: a periodic model (and a simple way to analyze it) is proposed in [1], an improvement with a strictly better utilization bound is given by [2]. The multiframe model [3] allows more precise analysis that suites well for video encoding tasks, and has been generalized to take deadlines into account by [4]. The recurring real-time task [5] and the digraph [6] models are more complex and can generalize the multiframe model, but at the cost of a higher complexity of the analysis. The method proposed in [7] allows determining if a system, composed of different types of tasks, is schedulable or not, only knowing the demand-bound function for each task. Another method proposed in [8], known as Response Time Analysis (RTA), provides a way to compute the exact worst-case response time for every task of a system.

ROS is one of the most used middleware in robotics and was already studied by [9] who shows the advantages of using ROS2 on industrial systems and [10] who looks at its real-time capabilities. Investigation on the influence of hardware and OS on ROS2's real-time performances was done in [11]. One remaining problem with using ROS for real-time systems is its communication model based on publish/subscribe mechanism that uses reactive execution model. The method proposed in [12] is very effective for analyzing chains of reactive tasks but assumes a *provided event arrival curve*, which is a big limitation. The method presented in [13] represents the system as a periodic pipeline of tasks; this approach works well when the event source that initiates the pipeline is periodic, which is an assumption we want to remove to be able to work with an unknown event source. The problem of event synchronization was addressed by [14], but for very complex systems and so the analysis is also complex and not very efficent on simple systems. The communication between tasks was handled by [15], but the focus was on multi-core architectures rather than on distributed architectures where network interferences have to be taken into account. A common way to make a reactive system analyzable is to use active polling, this way the activation of tasks is limited to known instants and the system is so deterministic. The method of [16] which proposes to model tasks using state machines can be used to study such tasks, but this would yield a model too complex to be verified.

In this paper, we present a method to represent reactive tasks with a deterministic execution model and provide solutions to analyze it. Section II introduces and formalizes our model for reactive tasks. Then, in section III, this model is analyzed, formulas are given for specific configurations and an algorithmic solution is presented for the general case. Section IV compares our algorithm with the resolution using an SMT solver. Finally, section V presents an implementation of our model in a real-time patch for ROS2.

II. Task and System Models

The objective is to analyze systems composed of periodic and reactive tasks, so a model compliant with the periodic model is needed for reactive tasks. As these hypotheses are valid in most of the recent systems, a first assumption considers a preemptive scheduler with fixed priority. Another assumption is that the tasks under analysis are statically allocated among CPUs with no migration at run-time and are independent, so a CPU per CPU analysis is sufficient and can be done on many-core architectures.

A. System

In the scheduling analysis context, a system S can be considered as a set of tasks, which will each generate an ordered sequence of jobs that will be executed. A task is characterized by its priority which defines the order in which tasks will be executed, and its execution parameters, which depend on the type of task and define the possible sequences of jobs. A periodic task $\tau_k \langle C_k, T_k \rangle$ is described by its period T_k and its worst-case execution time C_k and will generate every T_k unit of time a job that requires at most C_k unit of CPU time, a polling task $\rho_k \langle C_k^P, T_k^P, C_k^R, T_k^R \rangle$ will generate a sequence of jobs as described in II-B.

B. Polling Task

In this paragraph, the aims is to find a new model for reactive tasks that has similar characteristics in terms of reactivity, but which is deterministic and easily analyzable. The *polling task* model relies on active polling to ensure deterministic behavior. The state diagram of the polling model is presented in Fig. 1. At each iteration the polling state, in charge of verifying if a message has arrived, is executed for C^P units of time, then if there is no message the task goes to a waiting state for the polling period T^P. If there is a message, the task goes in the running state and process the callback for C^C units of time and then goes in a waiting state for the execution period T^R units of time. These characteristics are static. This model with two periods allows controlling individually the latency, i.e. the time between the reception of a message and its processing, and the CPU load. The polling period T^P fixes the time between two polling phases and so the latency, the execution period T^R limits the frequency of execution of the callback and so the CPU load generated by the task. By writing P the execution of the polling loop and R the execution of the running loop, traces of execution can be written in the form $[P, R]^*$. Figure 2 shows the trace of an execution that could be written P^3RP^2RP. To make easier the writing of equations in the rest of this paper, we will write:

$$C^R = C^P + C^C \tag{1}$$

This allows to see the task as two independent loops. A polling task is defined by giving the characteristics of the two loops:

$$\rho_k \langle C_k^P, T_k^P, C_k^R, T_k^R \rangle \tag{2}$$

with $C^R > C^P$.

The utilization factor of each loop and the global utilization factor are written:

$$U_k^P = \frac{C_k^P}{T_k^P};\ U_k^R = \frac{C_k^R}{T_k^R};\ U_k = \max(U_k^R, U_k^P) \tag{3}$$

This model is a generalization of the classic periodic model, with a polling state that only returns true the execution loop executed at each iteration and the task is so periodic. An example of polling task is the handling of communication with a GNSS, if the GNSS sends messages every $50ms$ with a precision on the frequency of $50\mu s$, assuming the time for computation of the message is $1ms$ and the time for executing the polling state is $5\mu s$, the polling tasks that implement this could have the following characteristics $\langle 5, 25, 1000, 50000 \rangle$.

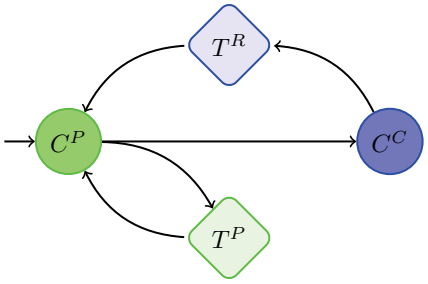


Fig. 1. Polling task state diagram. Round states are execution that lasts the held value, diamond states wait to the beginning date of the curent execution plus the held value.

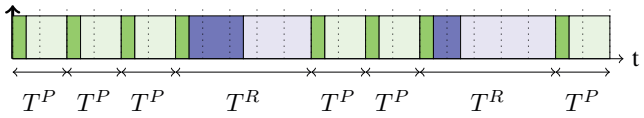


Fig. 2. Example of a polling task execution. Colors correspond to Fig. 1

III. Polling Task Analysis

Systems are composed of different types of tasks and even if it is possible to use the same model for every task, this will come at the cost of performance of the analysis or over-sizing the hardware part of the system. It is therefore interesting to be able to analyze them using a generic method that can be used among different types of tasks, in particular with periodic tasks which are the most often used.

A. Analysis principle

Two interesting ways of analyzing systems of tasks are the use of an utilization bound and the computation of the exact response time of each task. Utilization bounds, as introduced by Liu and Layland [1], are useful because they allow determining if a system is schedulable with a linear time but are not very precise and lead to reject systems that are schedulable. The response time analysis, although it is more complex, allows computing the exact worst-case response time

of all tasks of the systems and thus determining precisely if they all meet their deadline and so if the system is schedulable or not. It is, therefore, useful to be able to analyze a polling task with utilization bounds, but also with the response time analysis. The response time analysis, presented by [8], consists for periodic tasks in finding the fixed point of the sequence given by Eq. (4). The only prerequisite concerns the global utilization factor of the system $\sum U$ that must be lower than one.

$$\begin{aligned} \mathcal{R}^0 &= C_i \\ \mathcal{R}^{n+1} &= \left\lceil \frac{\mathcal{R}^n}{T_i} \right\rceil \cdot C_i + \sum_{j \in hp(i)} \left\lceil \frac{\mathcal{R}n}{T_j} \right\rceil \cdot C_j \end{aligned} \tag{4}$$

This method can be generalized for other types of tasks with Eq. (5), requiring only the knowledge of the release-bound function, defined as the maximum workload that a task can generate on the interval $[0, t[$, for each task of the system.

$$\begin{aligned} \mathcal{R}^0 &= \begin{cases} C_i & \text{for periodic tasks} \\ max\left(C_i^P, C_i^R\right) & \text{for polling tasks} \end{cases} \\ \mathcal{R}^{n+1} &= rbf_i(\mathcal{R}^n) + \sum_{j \in hp(i)} rbf_j(\mathcal{R}^n) \end{aligned} \tag{5}$$

With rbf_k the release-bound function of task τ_k and $hp(k)$ the set of tasks of equal or higher priority than τ_k. Thus, to analyze systems containing polling tasks with RTA it is necessary and sufficient to provide a way of computing the release-bound function of those tasks.

The rest of this section present a way to compute a linear upper bound of the rbf to allow an easy way of analyzing a system. Then, a more precise analysis of the polling task model is done. This analysis brings out a specific case and the corresponding analytical formulas to compute rbf. Next, a Sat Modulo Theory (SMT) formalization of the polling model allows handling the calculation of the rbf for the general case. Finally, a more efficient algorithm is presented.

B. Polling task request bound function

The release bound function of a polling task is a staircase function with different types of steps. The building of the curve uses two sizes of steps, one for the polling loop, and the other for the running loop. At the end of a step, i.e. the end of an iteration, the task is in its initial state and booth loops can be executed, this will generate two new possible steps that will each generate two other steps. Then the rbf is given by keeping for each t the higher step. This process of construction is shown in Fig. 3.

C. Request bound function upper bound

Because the rbf is complex to manage, it is useful to have an approximation that is easier to use during design phases when it is needed to have a quick way to find out temporal area in which the rbf will be. The loop with the higher utilization factor will drive the general shape of the rbf, the other loop will only generate interference. As shown on Fig. 3 the rbf will so follow a linear growth given by the global utilization

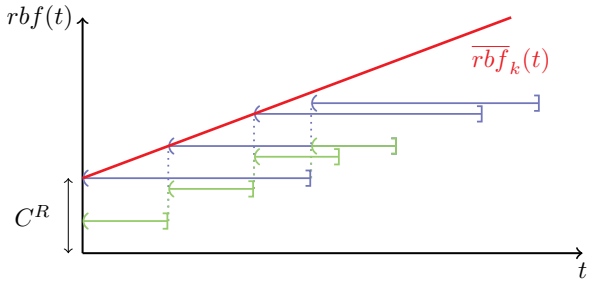


Fig. 3. Polling task request bound function with polling steps in green, running steps in blue and linear upper bound in red.

factor of the task, U_k, and the maximal interference is the given by the higher step, C^R. The upper bound is so given by Eq. (6).

$$\overline{rbf}_k(t) = U_k \cdot t + C_k^R \tag{6}$$

This linear bound allows computing the rbf with an $O(1)$ complexity. It is a pessimistic approximation, so systems found schedulable with the bound will always be schedulable, a less pessimistic bound is given by the staircase function of Eq. (7), this bound is not linear but still can be computed in $O(1)$.

$$rbf_k(t) = \lfloor U_k \cdot t \rfloor + C_k^R \tag{7}$$

D. Specific cases

When analyzing the curve, three phases are visible. The first lasts T_R and has an analytic formula, the third is periodic and so easy to analyze. In some conditions, there is a more complex phase between the two previous, it comes from the interweaving of different curves and lasts a duration that is bound but not fixed. Figure 4 gives an example of the rbf of a polling task.

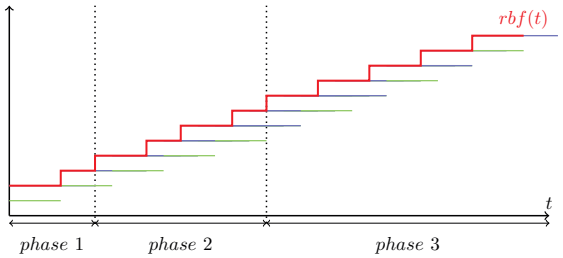


Fig. 4. Curve structure. Only steps that contribute to the rbf are shown.

a) Phase 1: The first phase lasts from $t = 0$ to $t = T_R$ the end of the running period, during this time the running loop can run only once. So the maximum CPU utilization is obtained when the polling loop runs a maximum times in $[0, t[$, and then the last iteration of the task is the running loop.

b) Phase 2: This phase is the most complex. The curve is the continuation of *phase 1* curve inter-weaved with other curves from executions that were dominated in *phase 1* but, in

phase 2, overtake locally the *phase 1* curve. This makes this part chaotic and prevents finding an analytic formula. In some conditions, this phase does not exist (see III-D3).

c) Phase 3: In this last phase, a pattern is repeated periodically, this pattern depends on the end of *phase 2* and so no generic formula can be found. When there is no *phase 2* it is *phase 1* which is repeated and so a formula can be found.

The analysis of the three phases allows us to point out some particular cases for which the rbf can be described using an analytical formula and so can be computed in O(1) complexity.

1) Running loop domination: When the polling period is greater than the running period, because $C^R > C^P$ the running loop dominates the polling loop, and, as it is shown in Fig. 5, any execution that contains one polling loop will have a release function lower than the release function of the 'only execution loop' execution, so the rbf is given by:

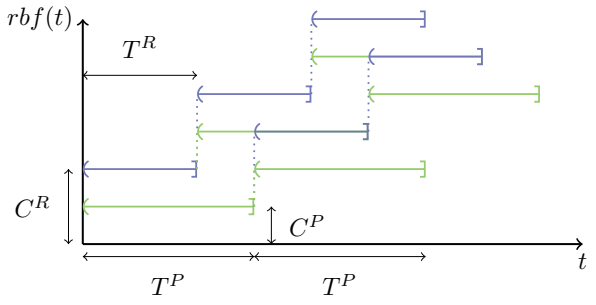


Fig. 5. Domination of the running loop

$$rbf(t) = \left\lceil \frac{t}{T^R} \right\rceil \cdot C^R \text{ iff } T^P \geq T^R \tag{8}$$

2) Phase 1: This phase can always be described with an analytical formula. The running loop can run only once, so every execution has the form $P^n R$, and the execution with maximum released CPU time is the one with the most polling loops and then one running loop, rbf is given by 9.

$$rbf(t) = \begin{cases} 0 & \text{if } t = 0 \\ \left(\left\lceil \frac{t}{T^P} \right\rceil - 1\right) \cdot C^P + C^R & \text{iff } t \in]0, T_R] \end{cases} \tag{9}$$

3) Simple harmonic case: When the running period is a multiple of the polling period every execution that did not take part in maximizing the rbf in phase 1 will overlap a curve that leaded to maximizing the rbf in *phase 1*, so no new interference is generated after T^R and *phase 2* does not exist. When the utilization factor of the polling loop is greater than the one of the running loop, the polling loop will dominate the execution, and every execution that uses the running loop will not leads to maximizing the rbf anymore. The execution that maximizes the released CPU time is the one using the most polling loops and then uses one running loop. The following formulas represent this "simple harmonic case":

$$T^R = k \cdot T^P \quad \text{with} \quad k \in \mathbb{N}^+ \quad \text{and} \quad U^P \geq U^R \tag{10}$$

$$rbf(t) = \begin{cases} 0 & \text{iff } t = 0 \\ \left(\left\lceil \frac{t}{T^P} \right\rceil - 1\right) \cdot C^P + C^R & \text{iff } t > 0 \end{cases} \tag{11}$$

4) Complex harmonic case: If the utilization factor of the running loop is greater than the one of the polling loop, the formula is more complex. In that case, the global shape of the rbf is given by the running loop and the pattern of *phase 1* is repeated on every step of the running loop. The formula is built by shifting right the equation of *phase 1* by the number of running periods in $[0, t[$ and adding the same number of C^R :

$$T^R = k \cdot T^P \quad \text{with} \quad k \in \mathbb{N}^+ \quad \text{and} \quad U^R \geq U^P \tag{12}$$

$$rbf(t) = \begin{cases} 0 & \text{iff } t = 0 \\ \left(\left\lceil \frac{m}{T^P} \right\rceil - 1\right) \cdot C^P + \left(\left\lfloor \frac{t}{T^R} \right\rfloor + 1\right) \cdot C^R & \text{iff } t > 0 \end{cases} \tag{13}$$

$$\text{with } m = t - \left\lfloor \frac{t}{T^R} \right\rfloor \cdot T^R \tag{14}$$

E. Optimization problem

For the general case, *phase 2* exists and there is no analytical formula. However, the calculation of the rbf can be written as an optimization problem that consists, for every t, of finding the number of iterations of each loop that maximize the released CPU time. An execution is made of a series of loops, each one could be the running or the polling loop. The amount of released CPU time on an interval is the sum of the execution time of each loop that occurred during this interval, every executions path with the same number of each loop will request the same amount of CPU time regardless of the order of the loops. By writing $r(t)$ the amount of request CPU time on $[0, t[$, i the number of execution loops and j the number of polling loops in $[0, t[$ we have:

$$r = i \cdot C^R + j \cdot C^P \tag{15}$$

To find the release bound function, we need to maximize $r(t)$. For t which are a linear combination of the periods, it means to find (i, j) that respects (16) and maximizes (15).

$$t = i \cdot T^R + j \cdot T^P \tag{16}$$

For values that are not a linear combination of the periods, we have to transform (16) into (17) but this condition does not take into account the last loop, which has not finished its execution yet. This loop could be either the polling or the execution loop but to maximize $r(t)$ we should use the maximum released time which is C^R, so (15) becomes (18):

$$\begin{cases} x < t \\ x = i \cdot T^R + j \cdot T^P \end{cases} \tag{17}$$

$$r = i \cdot C^R + j \cdot C^P + C^R \tag{18}$$

Figure 6 shows a description of this optimization problem using SMT-lib2[1] [17]. This language was chosen because it

[1] http://smtlib.cs.uiowa.edu/

is both sufficiently expressive and easily readable for humans. To resolve this problem, the Z3[2] [18] solver which is known for its efficiency was used. It is important to notice that other solvers or modelization languages could be used.

This SMT model was used to prove the earlier formulas, by writing a SMT modelization of the formulas[3], like in Fig. 7 for the bound, it is possible to ask the solver to find a counter example with `(assert (> r b))`, if the result is `unsat` then there is no counter example and the formulas is proven. The formulas of phase 1 and the bound (Eqs. (6, 9)) was proved with any other condition, the formulas of the running domination and the harmonic cases (Eqs. (6, 8, 11, 13)) were proved for bounded intervals.

```
;---------- inputs ----------
(define-const Tp Int  11)
(define-const Cp Int   1)
(define-const Tr Int  17)
(define-const Cc Int   2)
(define-const t  Int 100)
;------------------------------
(define-const Cr Int (+ Cp Cc))
(declare-const i Int)
(declare-const j Int)
(define-const x Int (+ (* i Tp) (* j Tr)))
(define-const r Int (+ (* i Cp) (* j Cr) Cr))
;-----
(assert (< x t))
(assert (< 0 i))
(assert (< 0 j))
(assert (< 0 r))
;---------- optimize ----------
(maximize r)
(check-sat); returns sat; rbf(100) = 19
(get-value (r)) ; 19
(get-value (i)) ; 1
(get-value (j)) ; 5
(get-value (x)) ; 96
```

Fig. 6. SMT modelization and rbf computation for a polling task

```
(declare-const n Int)
(assert (= t (* n Tp)))
(define-const b Int (+ (* n Cp) Cr))
(assert (>= (* Cp Tr) (* Cr Tp)))
```

Fig. 7. SMT modelization of the bound formula

F. Algorithmic solution

The SMT modeling of the problem introduced in the previous section finds an exact solution to the request-bound function of the polling tasks. However, this solution suffers from an efficiency problem. Indeed, its computation time is not negligible, and this problem increases with t. Moreover, this calculation is performed many times during the complete analysis of the system.

In this section, we present an algorithmic solution to the problem of computing the request-bound function of a polling task. This solution is more efficient than the previous one because it takes into account the structure of the rbf described in III-B. Indeed, the latter has a form of "lasso path": it starts with a sequence composed of *phase 1* then *phase 2*; then a loop formed by *phase 3*. The algorithm we propose calculates an efficient representation of this "lasso path". Once this computation is done, the cost of computing a query for a given t becomes negligible, which makes the method very efficient for the complete analysis of the system. We will show this efficiency in the next section. Figure 8 represents the lasso structure, which is composed of a set of nodes $\mathcal{N} = \{n_0, .., n_N\}$ and edges $\mathcal{E} = \{e_1, .., e_{N+1}\}$. The first part of nodes represents the sequence noted $\mathcal{S}$, and the second part is the loop noted $\mathcal{L}$. Each node represents a part of the query function. A node contains a set of segments ($n_1 = \{(x_1, y_1), \dots (x_n, y_n)\}$). Each segment represents either the execution of the polling loop or the execution of a running loop. Each edge is a shift on the axes of the request bound function ($e_1 = (dx_1, dy_1)$). This shift is a *centering* of the next node on the next point of interest. The notion of centering will be detailed later.

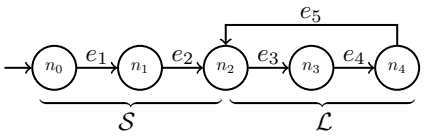


Fig. 8. Lasso representation

1) Computation principle: The first node contains the first execution of the polling loop and the first execution of the running loop ($n_0 = \{(x_0, y_0), (x_1, y_1)\}$).

The principle is as follows: we build the lasso structure step by step, following the rbf by moving from one point of interest to another. The next point of interest is the next (minimum t axis) end of the segments. At each iteration, all the new possible executions (polling and running) are added and filtered if necessary. When a new node is added we check if the same node has already been computed; in this case, the loop is detected and the lasso computation is complete, otherwise, the computation goes on. Algorithm 1 describes the computation of the lasso path. The centering function (Algorithm 2) computes the new edge while taking into account the new point of interest.

Segment drop: It is important to notice that it is not useful to add all the segments. Indeed, some segments are useless for the computation of the request bound function because it exists other segments that dominate the new one. Figure 9 highlight two cases in which the segment (x_j, y_j) is dominated by (x_i, y_i). Because the rbf is built recursively by adding the same two new segments to every point of interest, the curve originated from a point P_1 is the same as the curve originated from another point P_2 shifted by $\overrightarrow{P_2P_1}$, so if a segment ends later and lower than another segment the worst curve originated from the first will never overtake the one from the second.

Adding a new segment ($node.add$) can also lead to the dropping of dominated segments already in the current node.

[2]https://github.com/Z3Prover/z3
[3]https://gitlab.com/corail1/smt_proof

Algorithm 1 Lasso computation algorithm

```
1  function COMPUTE_LASSO
2      edges := ∅
3      nodes := ∅
4      node := [(T_P, C_P), (T_R, C_R)]
5      nodes.push(node)
6      loop
7          prev := nodes.last()
8          edge, node := prev.center()
9          edges.push(edge)
10         node.add((T_P, C_P))
11         node.add((T_P, C_P))
12         id = same_id(nodes, node)
13         if id then
14             sequence.nodes := nodes[1..id]
15             sequence.edges := edges[1..id]
16             cycle.nodes := nodes[id..len(nodes)]
17             cycle.edges := edges[id..len(nodes)]
18             return (sequence, cyle)
19         else
20             nodes.push(node)
```

Algorithm 2 Centering algorithm (used in Alg. 1)

```
1  function NODE::CENTER
2      first := self.first()
3      edge := (first.x, first.y)
4      new_node := ∅
5      for seg in self do
6          new_node.add((seg.x − edge.x, seg.y − edge.y))
7      return (edge, new_node)
```

Table I details all the dropping conditions. It is also important to notice that thanks to the dominated segments dropping, the next point of interest (next center) is unique.

TABLE I
SEGMENT DROP CONDITIONS

	$x_i < x_j$	$x_i = x_j$	$x_i > x_j$
$y_i < y_j$		drop (x, y_i)	drop (x_i, y_i)
$y_i = y_j$	drop (x_j, y_j)	drop one	drop (x_i, y_i)
$y_i > y_j$	drop (x_j, y_j)	drop (x_j, y_j)	

Computation example: Figure 10 highlights the first steps of the lasso structure computation in a simple example. Fig. 10a represent the initial node $n_0 = \{(x_0, y_0), (x_1, y_1)\}$ with $x_0 = T^P$, $y_0 = C^P$, $x_1 = T^R$, and $y_1 = C^R$. Then in this example, the next point of interest is the end of the polling loop (x_0, y_0). Fig. 10b show the new node centered with according to the first edge $e_1 = (x_0, y_0)$ in which the first segment is consumed by the centering function and the second is translated ($x'_1 = x_1 - x_0$, $y'_1 = y_1 - y_0$). The second node is computed (Fig. 10c) and then the third one (Fig. 10d).

2) Request bound function from Lasso: Once the lasso structure is computed, the calculation is straightforward. Let's notice $X(i)$ (resp. $Y(i)$) the sum of the dx (resp. dy) of the edges to reach node n_i:

$$X(i) = \sum_{j \in 1..i} dx_j \tag{19}$$

This notation will be used with $\mathcal{S}$ (resp. $\mathcal{L}$) to represent the sum of the dx of edges between nodes of the sequence (resp. loop). Let's $I(t)$ be the node index of the node useful to compute $rbf(t)$:

$$I(t) = \{max_{i \in 0..N}\ i \mid X(i) < t\} \tag{20}$$

The request bound function has two different formulas: one if no loop is needed to reach t, and another one if some loops are needed.

No loop: No loop is needed if the last node "is after" t:

$$t < X(N) + dx_{N+1} \tag{21}$$

In this case, we just have to find the maximum y coordinate of the segments, before t, of the useful node:

$$rbf(t) = Y(I(t)) + \left\{max_{(x,y) \in n_i}\ y \mid X(I(t)) + x < t\right\} \tag{22}$$

With loop: With loops the principle is to represent $t = X(\mathcal{S}) + k \cdot X(\mathcal{L}) + t'$ with $k = \left\lfloor \frac{t - X(\mathcal{S})}{X(\mathcal{L})} \right\rfloor$ and $t' = t - k \cdot X(\mathcal{L}) - X(\mathcal{S})$. Then the release function is defined as follows:

$$rbf(t) = Y(\mathcal{S}) + k \cdot Y(\mathcal{L}) + Y(\mathcal{L}(t')) \tag{23}$$

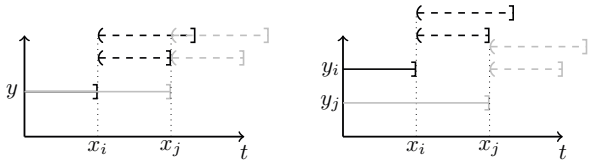
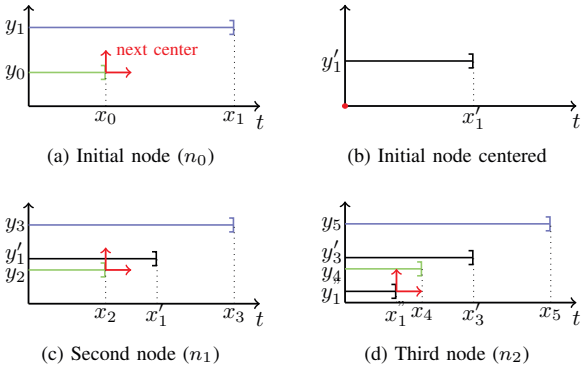


Fig. 9. Segment domination. Dashed segments represent the next step if the corresponding segment is kept



(a) Initial node (n_0)

(b) Initial node centered

(c) Second node (n_1)

(d) Third node (n_2)

Fig. 10. Lasso computation example. Blue and green segments are the two new segments added at the current step, black segments are remaining from previous step.

G. Analysis conclusions

Different ways of computing the rbf have been presented to suit the different situations. When a conservative approximation is sufficient a linear bound is given by Eq. (6). When an exact value of the rbf is wanted, if the situation corresponds to one of the cases in Tab. II the corresponding analytical formula can be used. In the general case the most efficient method is to use the lasso algorithm presented in section III-F.

TABLE II
SUMMARY OF THE SPECIFIC CASES

$T^R \leq T^P$		Eq. (8)
$\forall t \leq T^R$		Eq. (9)
$T^R = k \cdot T^P$	$U^R \leq U^P$	Eq. (11)
	$U^P \leq U^R$	Eq. (13)

IV. ALGORITHMS COMPARISON

Different methods to compute the release bound function for polling tasks are presented in this paper. This section presents the results of a benchmark that compares these methods in terms of performance. The aim of these algorithms is to be used in a more global algorithm, like the RTA, to compute the worst-case response time of a polling task in a complete system. In this context, the algorithm will be instantiated with a task, then the rbf will be computed for multiple values of t. In order to compare the performance of algorithms, we need to emulate these real conditions, so the benchmark will consist in creating a set of tasks, then for each algorithm, we measure for each task the time for instantiating the algorithm and compute the value of the $rbf(t)$ for different t. To mimic real systems, tasks were generated using random values for their parameters, worst-case execution times ($wcet$) are generated using a uniform distribution on $[0, 1000]$ and the periods with a uniform distribution on $[wcet, 100000]$.

Results

Tab. III shows the time for analyzing a system in function of the number of access to the rbf value for systems of different sizes. This is visible that the lasso algorithm is more efficient than solving the optimization problem with Z3. The experiments done shows that the lasso algorithm is more than 10^5 faster than the SMT solver. This can be explain by the way this two methods work. The optimization problem call the $maximize$ function of the SMT solver every time a value of the rbf is needed, this results in solving multiple optimization problem which are each complex to solve. On the other hand the lasso algorithm compute the lasso of Fig. 8 once, then accessing a value can be done by simply reading the lasso in a constant time.

TABLE III
COMPARISON OF ALGORITHMS

nb reading	5	10	15	30
Lasso (50 tasks)	10 ms	11 ms	12 ms	23 ms
SMT (50 tasks)	1.7e6 ms	2.8e6 ms	3.8e6 ms	8.6e6 ms

V. POLLING TASKS IMPLEMENTATION IN ROS2

Because ROS2 suffers from the impossibility to be analyzed since it widely uses reactive tasks, we develop an extension, Corail [4], which aims to solve this problem by providing an implementation using polling tasks.

[4]https://corail1.gitlab.io

A. ROS2

ROS is an open-source middleware dedicated to robotic applications and distributed systems which is based on the publish/subscribe paradigm. Elements of the system are called nodes and have to register to topics as a publisher or subscriber to be able to communicate. The version 2 was designed to be more real-time than the version 1. The use of DDS greatly improved the real-time capacity of the communication but the execution part still lack of real-time capabilities. The execution in ROS2 is handled by executors; the two default executors provided allow neither the preemption of tasks nor to order them by priority. Furthermore reactive model is used for subscription which prevent real-time analysis.

B. Corail

Corail is a ROS2 package that aims to make ROS2 more suitable for real-time systems. Corail relies on the concept of executors of ROS2 and provides a new real-time executor that ensures the respect of the assumptions needed for real-time analysis. This executor allocates a POSIX thread for each task of the system, which makes the task preemptive and allows using the POSIX scheduler to handle priority between tasks. Moreover, the Corail executor uses the polling task model to implement subscriptions and services. The use of POSIX and the polling task model enable Corail to ensure the respect of the constraints needed for real-time analysis.

C. Comparison

To compare ROS2 and Corail, two systems were implemented using both plain rclcpp and Corail. The first presented in Tab. Va is made of two periodic tasks, the second, presented in Tab. Vb is made of one periodic task and one subscription. Trace of the executions were made using LTTng[5], Figs. 11 and 12 show the traces of the implementation using only ROS 2. We can observe that there is no synchronization at the beginning of the run-time and executions of τ_1 when τ_2 is running are missed and that there is no preemption. This leads to systems that do not respect their deadlines while the two systems are found schedulable using the scheduling theory. The traces from the Corail implementations, detailed in Figs. 13 and 14, show executions that follow the hypotheses (preemption, synchronization, priority) and so the prevision of the scheduling theory. This is due to the fact that the execution follow the hypothesis needed for the scheduling theory. Moreover the use of the polling task model allows to compute the exact worst case response time of the reactive task without any knowledge of the event source.

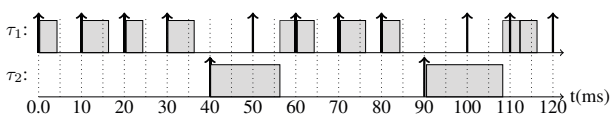


Fig. 11. Execution trace of system 1 with ROS 2

[5]https://lttng.org/

TABLE IV

	T	C	r	P
τ_1	**10**	**5**	0	1
τ_2	**50**	**15**	0	2

(a) System 1, two periodic tasks

	T	C	r	P
$\tau_1(periodic)$	**15**	**10**	0	1
$\tau_2(subscription)$	$\emptyset$	**10**	0	2

(b) System 2, one periodic task and one subscription

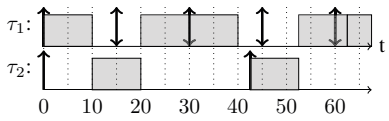


Fig. 12. Execution trace of system 2 with ROS 2

VI. Conclusion

In this paper, we address a problem that is regularly faced in our robotic experiments. Indeed, the embedded systems that we implement are reactive critical real-time systems. In this sense, they must be able to react to their environment but also communicate with each other while being analyzable from a real-time point of view. That is to say, it is essential to be able to show the respect of the deadlines of the different tasks. The difficulty lies in the fact that we have no guarantee either of communication times or of the dates of incoming events. To address this issue, we have proposed a method based on three pillars: a new task model that allows taking into account event polling, an analysis method that allows studying the respect of deadlines, and an implementation corresponding to this execution model (including a ROS2 patch).

We have given different solutions for the analysis method depending on the possible system configurations. Some have an analytical formula. For the general case, we have proposed a model in SMT that allows solving this problem and a specific algorithm. We explained the efficiency gain of the algorithmic version with the help of benchmarks. Finally, we presented our implementation in ROS2 for this new task model.

Our future work will first focus on the experimentation and deployment of these developments. Then we want to extend this execution model to take into account communication time because in certain configurations the networks we use allow it.

Acknowledgment

This work was supported by the Defense Innovation Agency (AID) of the French Ministry of Defense (research project CONCORDE N° 2019 65 0090004707501)

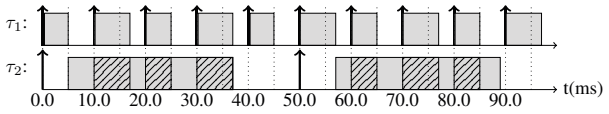


Fig. 13. Execution trace of system 1 with Corail, hatched areas correspond to preemption.

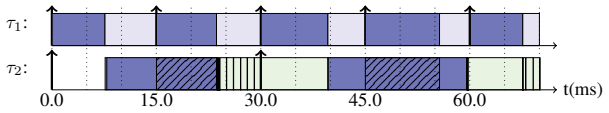


Fig. 14. Execution trace of system 2 with Corail, hatched areas correspond to preemption.

References

[1] C. L. Liu and J. Layland, "Scheduling algorithms for multiprogramming in a hard-real-time environment," *Journal of the Association for Computing Machinery*, vol. 20, no. 1, p. 46–61, Jan 1973.

[2] E. Bini, G. C. Buttazzo, and G. M. Butazzo, "Rate monotonic analysis: The hyperbolic bound." *EEE Transactions on Computers*, vol. 52, no. 7, pp. 933–942, 2003.

[3] A. Mok and D. Chen, "A multiframe model for real-time tasks," *IEEE Transactions on Software Engineering*, vol. 23, pp. 635–645, 1996.

[4] S. Baruah, D. Chen, S. Gorinsky, and A. Mok, "Generalized multiframe tasks," *Real-Time Systems*, vol. 17, no. 1, pp. 5–22, 1999.

[5] S. K. Baruah, "Dynamic-and static-priority scheduling of recurring real-time tasks," *Real-Time Systems*, vol. 24, no. 1, pp. 93–128, 2003.

[6] M. Stigge, P. Ekberg, N. Guan, and W. Yi, "The digraph real-time task model," in *2011 17th IEEE real-time and embedded technology and applications symposium*, Chicago, IL, 2011.

[7] I. Ripoll, A. Crespo, A. Mok, and J. Echagüe, "Improvement in feasibility testing for real-time tasks1," *IFAC Proceedings Volumes*, vol. 29, no. 6, pp. 205 – 212, 1996.

[8] M. Joseph and P. Pandya, "Finding response times in a real-time system," *The Computer Journal*, vol. 29, no. 5, pp. 390–395, 1986.

[9] E. Erős, M. Dahl, K. Bengtsson, A. Hanna, and P. Falkman, "A ROS2 based communication architecture for control in collaborative and intelligent automation systems," *Procedia Manufacturing*, vol. 38, pp. 349–357, 2019.

[10] L. Puck, P. Keller, T. Schnell, C. Plasberg, A. Tanev, G. Heppner, A. Roennau, and R. Dillmann, "Performance evaluation of real-time ROS2 robotic control in a time-synchronized distributed network," in *2021 IEEE 17th International Conference on Automation Science and Engineering (CASE)*, 2021.

[11] S. Sarkar, E. Lejeune, V. Lebastard, and A. Queudet, "Real-time jitter measurements under ROS2: the inverted pendulum case," 2021.

[12] D. Casini, T. Blaß, I. Lütkebohle, and B. B. Brandenburg, "Response-Time Analysis of ROS 2 Processing Chains Under Reservation-Based Scheduling," in *31st Euromicro Conference on Real-Time Systems (ECRTS 2019)*. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik GmbH, Wadern/Saarbruecken, Germany, 2019.

[13] G. Fohler and K. Ramamritham, "Static scheduling of pipelined periodic tasks in distributed real-time systems," in *Proceedings Ninth Euromicro Workshop on Real Time Systems*, Toledo, Spain, 1997.

[14] J. Garcia, J. Gutierrez, and M. Harbour, "Schedulability analysis of distributed hard real-time systems with multiple-event synchronization," in *Proceedings 12th Euromicro Conference on Real-Time Systems.*, Stockholm, Sweden, 2000.

[15] J. Fonseca, G. Nelissen, V. Nelis, and L. Pinho, "Response time analysis of sporadic dag tasks under partitioned scheduling," in *2016 11th IEEE International Symposium on Industrial Embedded Systems, SIES 2016 - Proceedings*. United States: Institute of Electrical and Electronics Engineers, 2016.

[16] N. Gobillot, C. Lesire, and D. Doose, "A Design and Analysis Methodology for Component-Based Real-Time Architectures of Autonomous Systems," *J Intell Robot Syst*, vol. 96, no. 1, pp. 123–138, 2019.

[17] C. Barrett, P. Fontaine, and C. Tinelli, "The SMT-LIB Standard: Version 2.6," Department of Computer Science, The University of Iowa, Tech. Rep., 2017, available at `www.SMT-LIB.org`.

[18] L. M. de Moura and N. S. Bjørner, "Z3: an efficient SMT solver," in *Tools and Algorithms for the Construction and Analysis of Systems, Hungary, March 29-April 6, 2008. Proceedings*, C. R. Ramakrishnan and J. Rehof, Eds., vol. 4963. Springer, 2008, pp. 337–340.