Analysis and Implicit Deadline Synchronization of Distributed Transactions Scheduled by EDF

Nicola Serreli, Giuseppe Lipari, Enrico Bini
Scuola Superiore Sant’Anna, Pisa, Italy
Email: {n.serreli,g.lipari,e.bini}@sssup.it

Abstract—In real-time distributed systems, it is common to model an application as a set of transactions, i.e., chains of tasks activated periodically, that must complete before an end-to-end deadline.

In this paper, (i) we extend the analysis of transactions scheduled by EDF to account for sporadic activations and (ii) we propose a protocol for assigning deadlines to tasks that does not rely on a global time reference.

First, we show that the scenario of strictly periodic activations is not the worst when the transactions are activated sporadically. For this reason we extend the demand bound function (dbf) to sporadic transactions and we propose a suitable schedulability analysis. Then, we propose IDSP (Implicit Deadline Synchronization Protocol) to assign the absolute deadlines to jobs at run time. The protocol does not require synchronization between nodes and uses only local information. We guarantee that the demand that can be generated at run time is always bounded by the sporadic dbf computed off-line.

I. INTRODUCTION

Distributed real-time systems are often modeled as a set of real-time transactions [1]. Each transaction is a chain of real-time tasks, and each task is allocated on a (possibly different) computational node. The first task in the transaction is activated periodically, or by external events characterized by a minimum interarrival time. The other tasks are activated upon the completion of the preceding one. All tasks in the transaction must complete within an end-to-end deadline relative to the activation time of the transaction. We allow the end-to-end deadline to be larger than the period. This situation is quite common in real applications. For example, in multimedia streaming, the period at which video frames are generated and processed may be lower than the end-to-end deadline for delivering the frames to the user.

An important problem is to check the schedulability of the system, i.e., to test if all transactions will complete before their end-to-end deadline under worst-case conditions. In fixed priority systems, the holistic analysis [1], [2] consists in reducing the overall distributed schedulability problem into $p$ single-node problems that can be solved using classical schedulability analysis. Each task is assigned a priority, and task parameters like offsets, jitters, response times are calculated so that the precedence constraints are automatically guaranteed. Since all schedulability problems depend on one another (i.e., the activation of an intermediate task, and hence its jitter, depends on the response time of the preceding task), the analysis is iterated until either a fixed-point solution is found or the set is deemed not schedulable. Similar techniques have been applied to EDF scheduling [3]–[6]. In this case, each task must be assigned an intermediate deadline instead of a fixed priority. Holistic analysis also allows to mix different schedulers on different nodes, as long as the designer is able to compute the worst-case response time of every task.

However, the holistic analysis is global, in the sense that it can be performed once the designer knows the parameters of all transactions. Moreover, any variation in one parameter (computation time, priority of an intermediate task,…) can influence the temporal behavior of all the system.

In a component-based approach, instead, it is desirable to perform the analysis in two steps: in the first step (local) we analyze each transaction in isolation, summarizing its temporal behavior with a (possibly small) set of temporal parameters. In the second step (integration), we must verify that the overall system is schedulable by only considering the temporal parameters derived in the first step.

Such an approach facilitates sensitivity analysis, increases robustness of the solution, allows to easily substitute one component with another, reduces the complexity of dynamic online admission control, etc.

In the local analysis it is not possible to use the response times of the tasks, because they depend on the presence of all other transactions. Therefore, in this paper we use the slicing approach [7]. Each task is assigned an execution window, and the execution windows of any two tasks of the same transaction are not overlapping. This can be done by assigning appropriate offsets and deadlines to every task.

Following the slicing method, under EDF the temporal characteristics of the transactions are summarized by a set of demand bound functions (dbfs) [8], one for each node. The integration analysis then consists on summing all the dbfs for every node, and check that the resulting function never exceeds the computational power of the node. In [9], we used such a methodology for periodic transactions, and we proposed a method for assigning intermediate deadlines to minimize a function of the dbfs.

However, two problems need to be solved to apply the slicing methodology to EDF. The first problem concerns with extending the analysis to sporadic transactions. In fact,
intermediate task deadlines based on the slicing approach
[9]. Our methods enables a component-based analysis.

Gantman et al. [15] presented a survey on synchronization
protoclos for real-time distributed systems. Among the many
algorithms presented in the survey, the Release Guard Proto-
col (originally proposed in [10]) achieves a smaller average
end-to-end response time, greatly reduces start-time jitter,
and does not require a global clock synchronization. The
protocol uses only local information regarding the minimum
separation time between instances of the same task, and
appropriately delays future instances so to guarantee that
higher priority tasks do not interfere too much with lower
priority tasks. However, the Release Guard Protocol only
works with fixed priority schedulers, and assumes that the
end-to-end deadline does not exceed the period. The protocol
has been enhanced by Zhang et al. [16] to deal with sporadic
transactions, again on fixed priority schedulers.

III. SYSTEM MODEL AND NOTATION

A distributed real-time application is modeled by a set of
transactions \( \{T_1, \ldots, T_n\} \). To simplify the presentation,
since our work investigates each transaction in isolation,
throughout the paper we drop the index of the transactions.

Transaction \( T \) is composed by a set of \( n \) tasks
\( \{\tau_1, \ldots, \tau_n\} \). Task \( \tau_i \), with \( i > 1 \), is activated upon
the completion of the preceding one \( \tau_{i-1} \) and it has a
computation time \( C_i \). The first task \( \tau_1 \) of the \( \ell \)th instance
of the transaction is activated at \( \Phi^\ell \), that is called absolute
activation. We denote by \( \tau_1^\ell \) the \( \ell \)th instance of the task \( \tau_i \).
We consider sporadic transactions with minimum iterarrival
time \( T \). Hence we have

\[
\Phi^\ell - \Phi^{\ell-1} \geq T. \tag{1}
\]

To describe a possible scenario of activations for the
sporadic transaction under analysis, we need to list the
possible values of absolute activations \( \Phi^\ell \). We label the
instance of the transaction under analysis by \( \ell \). Moreover,
we operate a time translation, so to set the activation of this
transaction at time reference \( 0 \). Therefore, we set \( \Phi^0 = 0 \).

The successive instances will be denoted by positive
indexes \( \ell > 0 \), and their absolute activations by \( \Phi^1, \Phi^2, \ldots \).
Similarly, the previous instances will be denoted by neg-
ative indexes \( \ell < 0 \), and their absolute activations by
\( \Phi^{-1}, \Phi^{-2}, \ldots \).

The following vector represents the sporadic activation
pattern:

\[
\Phi = (\Phi^{-k_0}, \ldots, \Phi^{k_1}) \tag{2}
\]

where \( k_0 \) and \( k_1 \) depend on the number of instances we need
to consider in the analysis (see Section V). Finally, \( \Gamma \) is the
set of all possible sporadic activation patterns.

We remark that, similarly to what it happens in multiplic-
cessor scheduling [17], activating the transactions as early
as possible (i.e. periodically) is not the worst-case for
the activation pattern. In Section V we show this by an example.
Each transaction $T$ has an end-to-end deadline $D$ that is the maximum tolerable time from the activation of the first task $\tau_1$ to the completion of the last task $\tau_n$. Since the analysis of the constrained deadline ($D \leq T$) is a straightforward extension of the classic analysis, throughout the paper we always assume $D > T$. In such a case, it may happen that a task is activated before its previous instance has completed. In this paper, we assume that the different activations of each task are served in a FIFO order.

The application is distributed across $p$ processing nodes, and each task $\tau_i$ of the transaction $T$ is mapped onto computational node $x_i \in \{1, \ldots, p\}$. Hence, we define $T_k = \{\tau_i \in T : x_i = k\}$ as the subset of tasks in $T$ mapped onto node $k$ and $n_k$ as the cardinality of $T_k$.

The delay due to network communication can be easily taken into account by considering the network as a special processing node, and messages as tasks. The methodology presented in this paper is valid also when different scheduling policies are used on the processing nodes. However, to simplify the presentation, in this paper we make two assumptions: we neglect the delay due to network communication (for example, restricting to a multiprocessor system with shared memory); and we assume EDF as the only scheduling algorithm in the system. A more general investigation will be presented in a future work.

Each task is assigned an intermediate deadline $\overline{D}_i$, that is the interval of time between the activation of the transaction and the absolute deadline of the task. Hence, using the notation introduced so far, the absolute deadline of the $i^{th}$ instance of $\tau_i$, is
\[
df{i} = \Phi^i + \overline{D}_i.\quad (3)
\]

We enforce the precedence relationship between tasks by the slicing technique [7]: for each task we set the activation offset $\phi_i$, relative to the activation of the transaction $\Phi^i$, equal to the intermediate deadline of the preceding one:
\[
\phi_1 = 0, \quad \phi_i = \overline{D}_{i-1} \quad i = 2, \ldots, n \quad (4)
\]

Clearly, the task absolute activation is
\[
o^i = \Phi^i + \phi_i. \quad (5)
\]

Moreover, we define the task relative deadline $D_i$ as
\[
D_i \triangleq \overline{D}_i - \phi_i.
\]

The relationship between activation offsets and relative deadlines is depicted in Figure 1. Clearly,
\[
\sum_{i=1}^{n} D_i = D \quad (6)
\]

The values of $T, \Phi^i, D, C_i, \overline{D}_i, D_i, \phi_i$ are all real numbers. Finally, we use the notation $\langle \cdot \rangle_0 \triangleq \max\{0, \cdot\}$.

IV. PERIODIC DEMAND BOUND FUNCTION

First, we recall the concept of demand bound function for a transaction that is strictly periodic (i.e. $\forall \ell, \Phi^\ell = \ell T$). Then, in the next section we extend the demand bound function to the sporadic case.

The computational requirement of the subset $T_k$ of tasks allocated on node $k$ is modeled by its demand bound function (dbf).

Definition 1: The demand function on node $k$, denoted by $df_k(t_0, t_1)$, is the total computation time of all the instances of the tasks in $T_k$, having activation time and deadline within $[t_0, t_1]$.

For periodic transaction, the demand function can be computed as follows [8]:
\[
df_k(t_0, t_1) \overset{df}{=} \sum_{\tau_i \in T_k} \left( \left\lfloor \frac{t_1 - \overline{D}_i}{T} \right\rfloor - \left\lfloor \frac{t_0 - \phi_i}{T} \right\rfloor + 1 \right) C_i \quad (7)
\]

As suggested by Rahni et al. [6], the overall demand bound function of $T_k$ in an interval of length $t$, is defined as:
\[
\text{dbf}_k(t) \overset{df}{=} \max_{t_0} \text{df}_k(t_0, t_0 + t) \quad (8)
\]

A necessary and sufficient schedulability test for non-concrete transactions (i.e. periodic transactions with free initial offset), scheduled by EDF consists in checking that the demand never exceeds the length of the interval on every processor
\[
\forall k = 1, \ldots, p \quad \forall t > 0 \sum_{T} \text{dbf}_k(T, t) \leq t \quad (9)
\]

where the sum is made over all the transactions in the system, and $\text{dbf}_k(T, t)$ denotes the demand bound function of $T$ on node $k$. In this case, first the dbf is computed for each transaction and for each node (applying the max operator), and then we sum all the dbf together to compute the overall computational requirement on node $k$.

In Figure 2 we illustrate the definitions introduced in this section by an example. Consider a transaction whose parameters are: period $T = 5$, end-to-end deadline $D = 8$, task deadlines $D_1 = 2$ and $D_2 = 6$, computation time $C_1 = 1$ and $C_2 = 3$. Both tasks are assigned to a single node. In the lower part of Figure 2, we show three consecutive instances of the transaction on three different nodes.
lines. In the upper part, we show the values of 3 functions: the demand in \([0, t]\); the demand in \([2, 2+t]\); and the demand bound function. We represent the points where the \(dbf\) has a step by a thick dot. The steps are tightly related to task deadlines. For example in the figure, the points \(p_1, p_2, p_3\) depend on the deadlines of task \(\tau_1\), while the points \(p_4, p_5\) depend on the deadlines of \(\tau_2\).

To compute the \(dbf\) of a periodic transaction, it is sufficient to consider the value of the demand functions obtained on the intervals that start with the activation of a task, as shown in [6]. Also, the \(dbf\) has a periodic pattern: its value for a generic large interval \(t\) can be computed as \(dbf(t') + jC\), where \(C = \sum_{\tau_i \in T_k} C_i, j \geq 0\) and \(t' = t - jT\) (see Section 4.1 in [6]).

V. SPORADIC DEMAND BOUND FUNCTION

Unfortunately, for sporadic transactions, the worst case does not occur with periodic activations. Consider the following transaction with 3 tasks on 2 processors. The transaction has period \(T = 5\) and end-to-end deadline \(D = 12\). The task parameters are reported in Table I.

![Figure 2: Example of demand bound function.](image)

![Figure 3: Example of sporadic transaction.](image)

<table>
<thead>
<tr>
<th>Task</th>
<th>(C_i)</th>
<th>proc.</th>
<th>(D_i)</th>
</tr>
</thead>
<tbody>
<tr>
<td>(\tau_1)</td>
<td>1</td>
<td>0</td>
<td>3</td>
</tr>
<tr>
<td>(\tau_2)</td>
<td>3</td>
<td>1</td>
<td>4</td>
</tr>
<tr>
<td>(\tau_3)</td>
<td>3</td>
<td>0</td>
<td>5</td>
</tr>
</tbody>
</table>

Table I: Parameters for the example

In Figure 3, we show two possible activation patterns. The first one corresponds to a periodic activation (\(\Phi^1 = T\)): in this case, it is easy to see that the maximum demand on processor 0 in any interval of length 5 is at most 3 units of computation.

In the second activation pattern we delay the activation of the second instance by 2 units of time (\(\Phi^1 = T + 2\)). As a consequence, the demand in interval \([7, 12]\) becomes 4 units of time, because one extra instance of \(\tau_1\) enters the interval. Thus, delaying an instance can increase the demand.

Hence, the analysis based on the classic periodic demand bound function is not applicable if transactions are sporadic. One of the contributions of this paper is to extend the demand bound function to sporadic transactions.

A job \(\tau_i^\ell\) in \(T_k\), runs inside interval \([t_0, t_1]\) if its absolute deadline \(d_i^\ell\) is not smaller than \(t_1\)

\[ t_1 \geq d_i^\ell = D_i + \Phi^\ell \tag{10} \]

and its activation is not earlier than \(t_0\)

\[ t_0 \leq a_i^\ell = \phi_i + \Phi^\ell \tag{11} \]

By introducing the function

\[ \text{step}(x) = \begin{cases} 
0 & \text{if } x < 0 \\
1 & \text{if } x \geq 0 
\end{cases} \tag{12} \]

we can define the following binary-valued function

\[ \text{jobln}_i(t_0, t_1) \overset{\text{def}}{=} \text{step}(t_1 - D_i - \Phi^\ell) \cdot \text{step}(\phi_i + \Phi^\ell - t_0) \tag{13} \]

that returns 1 if the job \(\tau_i^\ell\) has both activation and deadline in \([t_0, t_1]\), and it returns 0 otherwise.

Hence, the demand of all the tasks belonging to the transaction \(T_k\) can be expressed as:

\[ \text{df}_k(t_0, t_1) \overset{\text{def}}{=} \max_{\Phi \in \Gamma} \sum_{\ell = -k_0}^{k_1} \sum_{\tau_i \in T_k} \text{jobln}_i(t_0, t_1) C_i \tag{14} \]

where \(k_0\) and \(k_1\) are indexes of transaction instances (later determined in Eq. (18)) that may have an effect on the demand in \([t_0, t_1]\).

The sum on all the transaction instances \(\ell\) can be split in three parts: the first part is the sum over the indexes corresponding to the past instances (from \(-k_0\) to \(-1\)); the second part is the current instance (with \(\ell = 0\)), and the third part is the sum over the future instances (from 1 to
\[ k_1 \). Hence Equation (14) becomes
\[
df_k(t_0, t_1) = \max_{\tau_i} \left( \sum_{\ell=-k_0}^{+1} \phi_i(t_0, t_1) C_{\tau_i} + \sum_{\ell=k_1}^{+1} \phi_i(t_0, t_1) \right)
\]
\[
= \sum_{\tau_i} \ldots + \max_{(\phi_1 \ldots, \phi_{k_1}) \in \Gamma^+} \sum_{\ell=-k_1}^{+1} \phi_i \sum_{\ell=1}^{+1} \tau_i \ldots
\] (15)

where \( \Gamma^- \) and \( \Gamma^+ \) are the sets of the possible activation patterns of the past and the future instances respectively. Although Eq. (15) is apparently more complex than Eq. (14), it will be more useful for our purposes because it has the advantage of decoupling the dependence on past and future instances (see Sections V-A and V-B).

Finally, as for the periodic dfb, the sporadic dfb is the maximum among all the sporadic demand functions computed on intervals with the same length:
\[
dfb_k(t) \triangleq \max_{t_0} df_k(t_0, t_0 + t) \] (16)

Figure 4 shows that, for the same parameters of Table I, the sporadic dfb computed from Eq. (16) is larger than the periodic dfb (Eq. (8)).

Figure 5: Algorithm for computing the dfb.

Section V-A we describe the procedure STOREINTERVALS for performing this step. After sorting the intervals \([t_0, t_1]\) in intervalSet by increasing \(t_1 - t_0\) (at line 3), we search for the activation pattern \(\Phi\) that maximizes the demand in \([t_0, t_1]\). In Section V-B we describe the procedure SPANPATTERNS that computes the set \(\Gamma\) of all possible activation patterns.

A. Enumerating the intervals

The first stage requires to enumerate all the intervals \([t_0, t_1]\). The pseudocode of this stage is reported in Figure 6. First, we claim that \(t_0\) must coincide with the activation of some job. In fact, if this does not happen then we could increase \(t_0\) achieving a shorter interval with the same demand. Hence we set \(t_0\) equal to the activation of the job \(\tau^0\), i.e. \(t_0\) spans on \(\{\phi_i : \tau_i \in T_k\}\) (see line 4 of the algorithm). Notice that, without loss of generality, we label by \(0\) the transaction instance which this job belongs to.

Regarding the possible values of \(t_1\), it is easy to see that it is sufficient to test only the absolute deadlines \(d_j\). In fact if \(t_1 = d_j\) for some task \(\tau_j \in T_k\) and some transaction instance \(h\), then a reduction of \(t_1\) by an arbitrary small amount \(\varepsilon\) will decrease the demand \(df\) by at least \(C_j\). However, the main difficulty here is that the absolute activations are not fixed, hence we do not know where the deadlines are until we fix the sporadic activation pattern \(\Phi\).

First, we list the values of \(t_1\) associated with the absolute deadlines of the instance 0 (see lines 5–9). Then we invoke the recursive procedures FUTUREDEADLINE and PASTDEADLINE that list the absolute deadlines of the future and past instances, respectively.

These two procedures explore the possible activation patterns \(\Phi\) such that task activations are aligned with \(t_0\). For each pattern the values of absolute deadlines are recorded as candidate values for \(t_1\).

We conclude the section by showing that, after a transient that is long at most \(D + T\), the dfb becomes periodic.
procedure \texttt{STOREINTERVALS}  
1: \texttt{intervalSet} \leftarrow \emptyset \quad \triangleright\text{initialize}  
2: \textbf{for} each $\tau_i \in T_k$ \textbf{do} \quad \triangleright\text{loop on} t_0  
3: $t_0 \leftarrow \phi_i$  
4: $\textbf{for} \tau_j \in T_k \textbf{do}$  
5: \textbf{if} $\mathcal{D}_j > t_0$ \textbf{then}  
6: \hspace{1em} \text{store} $[t_0, \mathcal{D}_j]$ in \texttt{intervalSet}  
7: \textbf{end if}  
8: \textbf{end for}  
9: \textbf{end procedure}  

procedure \texttt{FUTUREDEADLINE}($t_0$, $\ell$, $\Phi^{\ell-1}$)  
10: \textbf{for} all $\Phi^\ell \in \{\Phi^{\ell-1} + T \} \cup \{ t_0 - \phi_i : t_0 - \phi_i > \Phi^{\ell-1} + T, \tau_i \in T_k \}$ \textbf{do}  
11: \textbf{for} each $\tau_i \in T_k$ \textbf{do}  
12: $t_1 = \Phi^\ell + \mathcal{D}_i$  
13: \textbf{if} $t_1 > t_0$ \textbf{then}  
14: \hspace{1em} \text{store} $[t_0, t_1]$ in \texttt{intervalSet}  
15: \textbf{end if}  
16: \textbf{end for}  
17: \textbf{end for}  
18: \textbf{end procedure}  

procedure \texttt{PASTDEADLINE}($t_0$, $\ell$, $\Phi^{\ell+1}$)  
19: \textbf{for} all $\Phi^{\ell} \in \{\ell T, \Phi^{\ell+1} + T\} \cup \{ t_0 - \phi_i : \Phi^{\ell+1} + T < t_0 - \phi_i < \ell T, \tau_i \in T_k \}$ \textbf{do}  
20: \textbf{for} each $\tau_i \in T_k$ \textbf{do}  
21: $t_1 = \Phi^\ell + \mathcal{D}_i$  
22: \textbf{if} $t_1 > t_0$ \textbf{then}  
23: \hspace{1em} \text{store} $[t_0, t_1]$ in \texttt{intervalSet}  
24: \textbf{end if}  
25: \textbf{end for}  
26: \textbf{end for}  
27: \textbf{end procedure}  

Figure 6: Algorithm for enumerating intervals.

\begin{align*}
\forall t > D + T \quad \text{dbf}_k(t + T) = \text{dbf}_k(t) + C^k.
\end{align*}

where $C^k = \sum_{\tau_i \in T_k} C_i$.

\textbf{Proof:} Let $t_0$ and $\mathcal{T}$ be the instant and activation pattern that give the value of \text{dbf}_k(t) in Equations (16) and (15) respectively, and let us set $t_1 = t_0 + t$.

We identify with $\ell$ the first transaction instance with activation $\Phi^\ell > t_0$, hence $\Phi^{\ell-1} \leq t_0$. Since we are in the worst case and $\Phi^\ell > t_0$, then

\begin{align*}
\forall h \geq \ell \quad \Phi^h - \Phi^{h-1} = T
\end{align*}

otherwise, we could anticipate all $\Phi^h$ with $h \geq \ell$ without removing any job from the interval. On the contrary, the deadline of a job may enter the interval, and the worst-case activation pattern cannot be $\mathcal{T}$ anymore, causing a contradiction.

From (17) and the definition of $\ell$, we notice that the instance $\ell$ of the transaction ends earlier than $t_1$. Clearly this is also true for all instances before $\ell$. Formally

\begin{align*}
\Phi^{\ell-1} \leq t_0 & \quad \Rightarrow \quad \Phi^\ell \leq t_0 + T \\
\Phi^\ell + D \leq t_0 + T + D < t_1.
\end{align*}

From (17), it follows that any interval of length $T$ starting after $\Phi^\ell + D$ contains exactly one activation and one deadline of each task. Hence the demand generated in the interval $[t_0, t_1 + T]$ increases by one job for all tasks in $T_k$, i.e. $C^k$.

Suppose by absurd that $\text{dbf}_k(t + T) > \text{dbf}_k(t) + C^k$. Then, it exists an interval $[t_0, t_0' + t + T]$ with demand larger than $\text{dbf}_k(t) + C^k$. Let $\mathcal{T}$ be its activation pattern, and let us call $\ell'$ the first instance with $\Phi^{\ell'} > t_0'$. Followong the same reasoning as above, the demand in $[t_0', t_0' + t]$ decreases by $C^k$. However, this is absurd because we obtain a new interval with the same length $t$ but with demand higher than in $[t_0, t_0 + t]$.

Since, thanks to the lemma, the transient part of the dbf lasts for no longer than $D + T$ and the periodic part is long $T$, it is possible to compute the dbf only for lengths of intervals less than $D + 2T$.

Now we present an algorithm for computing the activation patterns that determines the maximum demand in a given interval $[t_0, t_1]$.

\textbf{B. Algorithm for enumerating the activation patterns}

In this section we explain the procedure \texttt{SPANPATTERNS}($t_0, t_1$) (see line 7 of the algorithm in Figure 5) that checks all possible sporadic activation patterns of past and future instances that may have an impact on the interval $[t_0, t_1]$. Therefore, we are interested only in transaction instances that may overlap with the interval $[t_0, t_1]$. The indexes of these transactions are from $-k_0$ to $k_1$, where

\begin{align}
k_0 & = \left\lceil \frac{D - t_0}{T} \right\rceil - 1 \quad k_1 = \left\lceil \frac{t_1}{T} \right\rceil - 1.
\end{align}

Hence the sum of transactions instances of Eq. (15) has to be made for $\ell = -k_0, \ldots, k_1$.

For the example of Table I (see also Figure 4 for a timeline representation of the instances), if we set $t_0 = \phi_1 = 0$ and
\[ t_1 = 13, \text{ we find } k_0 = 2 \text{ and } k_1 = 2, \text{ meaning that in the analysis of the demand in the interval } [5, 12] \text{ we consider the instances } -2(= -k_0), -1, 0, 1, 2(= k_1) \text{ of the transaction.} \]

In the exploration of the activation patterns we distinguish between future instances (with index \( \ell > 0 \)) and past instances (with index \( \ell < 0 \)). The guideline for the exploration of absolute activations of future instances is to align some task activation \( a^\ell_i = \Phi^\ell + \phi_i \) with \( t_0 \). This is possible by setting

\[ \Phi^\ell = t_0 - \phi_1. \]  
(19)

However, this is a valid absolute activation only if it respects the constraints of being a sporadic transaction with minimum interarrival \( T \), that is

\[ \Phi^\ell \geq \Phi^{\ell-1} + T. \]  
(20)

This condition introduces a recurrent dependency between all the values \( \Phi^0, \Phi^1, \Phi^2, \ldots, \Phi^{k_1} \). The procedure \textsc{computeFuture} for testing future instances is reported in Figure 6.

The same rationale is applied to past instances (the ones with index \( \ell < 0 \)). In this case however, we aim at finding the absolute activation \( \Phi^\ell \) such that some absolute deadline is aligned with \( t_1 \). The full algorithm that explores the activation patterns is reported in Figure 7.

In the example of Figure 4, if we assume \( t_0 = 0 \) then \( \Phi^1 \) should be tested with the values of 5(=\( T \)). Instead, if \( t_0 = \phi_3 = 7 \) then \( \Phi^1 \) is checked both when it is 5 and when it is \( t_0 - \phi_1 = 7 \), meaning that we align the activation of the instance 1 with the offset \( \phi_3 = t_0 = 7 \).

C. Complexity analysis

We start by analysing the complexity of procedure \textsc{storeIntervals}. The outer loop (line 3) is executed \( n_k \) times. After adding the intervals for instance 0, procedures \textsc{FutureDeadline} and \textsc{PastDeadline} are invoked.

Procedure \textsc{FutureDeadline} explores a number of instances at most equal to \( k_2 = \left\lceil \frac{D_{-2} - T_0}{T} \right\rceil - 1 \). Of this, the first \( \left\lceil \frac{D_{-2} - T_0}{T} \right\rceil \) instances may vary their activation time, while for the successive ones, the worst-case corresponds to interarrival times equal to \( T \). The number of possible combinations of activations (line 16) is then \( n_k \left\lceil \frac{D_{-2} - T_0}{T} \right\rceil \). For each combination, \( n_k k_2 \) deadlines are generated.

Procedure \textsc{PastDeadline} is very similar. The number of instances is \( k_0 \) (see Eq. (18)). The maximum number of elements generated for each combination of past activations is \( n_k k_0 \). Finally, the maximum number of combinations (line 29) is \( (n_k + 2)^{k_0} \).

Each generated interval must be inserted in a ordered list, an operation that takes logarithmic time in the size of the list. The size of the list at the end of the procedure is:

\[ s = k_2 n_k \left\lceil \frac{D_{-2} - T_0}{T} \right\rceil + 1 + n_k k_0 (n_k + 2)^{k_0} \]

and the complexity is \( O(\sum_{i=1}^{s} \log(i)) \).

1: procedure \textsc{SpanPatterns}(\( t_0, t_1 \))
2: \( k_1 = \left\lceil \frac{D_{-2}}{T} \right\rceil - 1 \) \hspace{1cm} \( \triangleright \) see Eq. (18)
3: \textsc{ComputeFuture}(1, (0, \ldots, 0)) \hspace{1cm} \( k_1 \)
4: \( k_0 = \left\lceil \frac{D_{-2} - T}{T} \right\rceil - 1 \) \hspace{1cm} \( \triangleright \) see Eq. (18)
5: \textsc{ComputePast}(-1, (0, \ldots, 0))
6: end procedure

7: procedure \textsc{ComputeFuture}(\( \ell, (\Phi^1, \ldots, \Phi^{k_1}) \))
8: if \( \ell > k_1 \) then \hspace{1cm} \( \triangleright \) the exit condition
9: \quad store \( (\Phi^1, \ldots, \Phi^{k_1}) \) in \( \Gamma^+ \) \hspace{1cm} \( \triangleright \) \( \Gamma^+ \) is global
10: else
11: \quad \( \Phi^0 \leftarrow 0 \)
12: \quad for all \( \Phi^\ell \in \{\Phi^{\ell-1} + T\} \cup \{t_0 - \phi_i : t_0 - \phi_i > \Phi^{\ell-1} + T, \tau_i \in T_k\} \) do
13: \quad \quad \textsc{ComputeFuture}(\( \ell + 1, (\Phi^1, \ldots, \Phi^{k_1}) \))
14: \quad end for
15: end if
16: end procedure

17: procedure \textsc{ComputePast}(\( \ell, (\Phi^{-k_0}, \ldots, \Phi^{-1}) \))
18: if \( \ell < -k_0 \) then
19: \quad store \( (\Phi^{-k_0}, \ldots, \Phi^{-1}) \) in \( \Gamma^- \)
20: else
21: \quad \( \Phi^0 \leftarrow 0 \)
22: \quad for all \( \Phi^\ell \in \{\Phi^{\ell+1} - T\} \cup \{t_1 - \tau_i : t_1 - \tau_i < \Phi^{\ell+1} - T, \tau_i \in T_k\} \) do
23: \quad \quad \textsc{ComputePast}(\( \ell - 1, (\Phi^{-k_0}, \ldots, \Phi^{-1}) \))
24: \quad end for
25: end if
26: end procedure

Figure 7: Algorithm for generating \( \Gamma^- \) and \( \Gamma^+ \).

Notice that, while generating the the values of \( t_1 \), it is quite common to obtain many times always the same values. In average, we expect that the final size of the list is much smaller than its upper bound \( s \).

Regarding procedure \textsc{SpanPatterns}, we apply a similar reasoning. We address separately future and past instances. Procedure \textsc{ComputeFuture} builds a tree in which at level 1 sets the value of \( \Phi^1 \), at level 2 sets the value of \( \Phi^2 \), and so on. There will be \( k_1 \) levels. Each node has at most \( n_k + 1 \) children. Hence, the number of leaves of such a tree is \( (n_k + 1)^{k_1} \). Each leaf corresponds to a different value of \( (\Phi^1, \ldots, \Phi^{k_1}) \). A similar tree can be built for past instances. Thus the complexity of enumerating all activation patterns is

\[ O((n_k + 1)^{k_0} + (n_k + 1)^{k_1}). \]
Finally, the complexity of computing the whole dbf is
\[ O\left( \sum_{i=1}^{k} \log(i) + s n_k \left( (n_k + 1)^{k_0} + (n_k + 1)^{k_1} \right) \right). \]

We are aware that the proposed algorithm is very complex. Most of the complexity lies in the sporadicity of the transaction that requires to check all possible scenarios. In this paper, we focused on the exact analysis regarding of its complexity. We leave to future investigations the development of simplified algorithms as well as Fully Polynomial Time Approximation Schemes (FPTAS).

VI. IMPLICIT DEADLINE SYNCHRONIZATION PROTOCOL

In order to implement a system that uses the transaction model presented in this paper, the scheduler on each node must set the activation times and the deadlines of the jobs so that, if all tasks execute for less than their worst-case execution times \( C_i \), and the difference between two consecutive transaction instances is greater than the minimum interarrival time \( T \),

1) every transaction always respects its end-to-end deadline;
2) in every interval, the sporadic demand of every transaction is always less than or equal to the sporadic demand computed off-line.

At first glance, it may seem that we need a strict clock synchronization protocol to guarantee that the activations and the deadlines are correctly computed. In fact, in a distributed system the timing information are obtained on each node by reading local timer hardware interfaces, and different timers can have different offsets and different speeds. Therefore, a global clock synchronization protocol is often used to synchronize the different timing views to the one taken as reference.

In this paper, instead, we show that it is possible to remove the need for a common time reference. However, we still assume that there is no drift among the timers of different nodes. We plan to extend our analysis to distributed systems in a future work.

Our idea is similar to the Release Guard Protocol (RGP) [10] by Sun and Liu. RGP has been thought for reducing the start-time jitter and guaranteeing a minimum separation time between two task activations in a fixed priority system. RGP works by delaying the activations of a task so that the distance between two consecutive instances is never less than the minimum interarrival time.

The algorithm we propose uses a similar idea to impose a minimum distance between deadlines.

We start by observing that, once we assign the correct deadline to a job, we can also let it start before its offset \( \phi_i \).

Lemma 2: Anticipating the activation of a task without modifying its absolute deadline does not increase the sporadic dbf.

Proof: Let \( \phi_i = \Phi^f + \phi_i \) be the activation of job \( \tau_i^f \), \( d_i^f \) its absolute deadline and let \( a_i^f < \phi_i \) be its actual starting time. Then, \( d_i^f - a_i^f > D_i \).

Let \( t > D_i \), and let \( t_0 \) be such that \( df(t_0, t_0 + t) \) is maximal. If the interval contains \( \phi_i \) and \( d_i^f \) but not \( a_i^f \), then the dbf may decrease. In all other cases, the dbf in \( t \) remains the same.

While for the purpose of the analysis we impose that the activation of a task is equal to the deadline of the previous task, thanks to Lemma 2 at run-time we can activate a job at the completion of the previous job, as long as the deadlines are correctly set.

The second observation is that the dbfs of different nodes are not related to each other. If we restrict our attention to a node \( k \), as long as the system is schedulable and all tasks meet their deadlines, we can use the activation of some task in \( T_k \) as a reference time to compute all other parameters.

Before describing the protocol, we need an additional definition.

Definition 2: We define as precedence set \( P_i^f \) of job \( \tau_i^f \) the set of all jobs \( \tau_j^f \) of tasks \( \tau_j \in T_k \), with \( h \in \{ \ell, \ell - 1, \ell - 2, \ldots \ell - k_0 \} \), that have absolute deadline less than \( d_i^f \) under all possible activation patterns. The number \( k_0 \) of instances to consider is:

\[ k_0 = \left\lfloor \frac{D}{T} \right\rfloor - 1. \]

The precedence sets impose a partial order on the set of jobs belonging to previous instances. In practice, the deadline \( d_i^f \) of job \( \tau_i^f \) must necessarily follow all the deadlines of jobs \( P_i^f \) under all possible activation patterns.

A stronger property is the following.

Lemma 3: At run time, the distance between \( d_i^f \) and any of the jobs in \( P_i^f \) cannot be less than

\[ d_i^f \geq d_j^f + (\ell - h)T + \overline{D}_i - \overline{D}_j \]

Proof: Follows directly from the definitions.

A. The protocol

We now present the IDSP algorithm to compute the absolute deadline of a job \( \tau_i^f \) at the instant of its activation \( a_i^f \). The protocol consists of three simple rules.

Rule 1 The separation between activation and deadline of \( \tau_i^f \) must always be greater than its relative deadline:

\[ d_i^f \geq a_i^f + D_i \]  \hspace{1cm} (22)

Rule 2 The second rule mandates a minimum separation between the deadlines of the jobs of the same task:

\[ d_i^f \geq d_i^{f-1} + T. \]  \hspace{1cm} (23)

Rule 3 The distance between \( d_i^f \) and any job in \( P_i^f \) must not be less than the minimum possible distance as computed off-line. In formula:

\[ \forall \tau_j^f \in P_i^f, \quad d_i^f \geq d_j^f + (\ell - s)T + \overline{D}_i - \overline{D}_j \]  \hspace{1cm} (24)
It may happen that a job \( \tau_i \) is activated before a job in its precedence set, due to the fact that the end-to-end deadline can be larger than period and previous jobs complete much earlier than their deadline. In such a case, the job is suspended and its deadline cannot be computed until we have computed the deadlines of all the jobs in its precedence set.

From the previous rules, the job deadline \( d_i^\ell \) can simply be computed as the maximum among the RHS the three inequalities. Notice that we only use parameters that are local to each node \((a_i^\ell, d_i^\ell)\) or statically known \((D_1, T, D_1, P_1, D_1)\) and the intermediate deadline \(D_j\) for each job \( \tau_j^h \in P_1^\ell \).

### B. Computing the precedence set

Set \( P_1^\ell \) can be very large. We will now show that it is possible to only compute a smaller subset \( P_1^\ell \) that contains at most one job per past instance. First of all, let us show how to compute such reduced subset:

- Set \( P_1^\ell \) contains at most one job per each instance \( \ell, \ell-1, \ldots, k_0 \). Initially, \( P_1^\ell \) is empty.
- Considering instance \( \ell = k \), we only need to know the task \( \tau_j \in T_k \) that immediately precedes \( \tau_i \) in the transaction. Then, \( \tau_j^\ell \) is added to \( P_1^\ell \). If \( \tau_i \) has no preceding task in \( T_k \), we skip this step.
- Then, we enter a cycle in which we compute the job for instance \( h \in \ell-1, \ell-2, \ldots, k_0 \). Let \( d_m \) be the greatest deadline among the jobs already in \( P_1^\ell \).
- Consider the latest job \( \tau_j^m \) that has absolute deadline in interval \((d_m, d^\ell_i)\), under the assumption of periodic distance between all instances from \( h \) to \( \ell \). If such a job exists, its deadline \( d_j^h \) is added to \( P_1^\ell \). Else, we skip to the next instance.
- We iterate until instance \( k = k_0 \).

An example of the procedure is shown in Figure 8. Consider a transaction having 6 tasks, with period \( T = 10 \) and end-to-end deadline \( D = 25 \). Hence, we need to consider \( k_0 = 2 \) instances. The intermediate deadlines are respectively, 3, 6, 10, 14, 21, 25. We assume that tasks \( \tau_1, \tau_3, \tau_5 \) are allocated on processor \( k = 1 \), while task \( \tau_2, \tau_4, \tau_6 \) are allocated on processor \( k = 2 \). We want to compute the precedence set of job \( \tau_2 \) (the second task in the bottom line of Figure 8). By setting all preceding activations at distance equal to \( T \), we have the activation pattern shown in the figure. The precedence set of \( \tau_2^\ell \) contains: 1) no job of instance \( \ell \), because there is not task preceding \( \tau_2 \) on processor 2; 2) job \( \tau_4^\ell-1 \); 3) job \( \tau_6^\ell-2 \).

According to rule 3, we must check that \( d_2^\ell \geq d_4^\ell-1 + 2 \) and \( d_2^\ell \geq d_6^\ell-2 + 1 \). Therefore, \( d_2^\ell \) can be computed as:

\[
d_2^\ell = \max\{a_2^\ell + 25, d_2^\ell-1 + 10, d_4^\ell-1 + 2, d_6^\ell-2 + 1\}.
\]

In general, the maximum number of elements to maximize is upper bounded by \( \min\{k_0, n_k - 1\} + 2 \), so it depends on the ratio between end-to-end deadline and period, and in no case is greater than \( n_k + 1 \).

### C. Proof of correctness

Rule 3 mandates that all the deadlines in the precedence set \( P_1^\ell \) must be computed before we can compute deadline \( d_i^\ell \). The following lemma proves that at run-time it is sufficient to only consider \( P_1^{\ell+} \).

**Lemma 4:** If all the jobs in \( P_1^{\ell+} \) have been assigned a deadline at run-time, then all the jobs in \( P_1^\ell \) have been assigned a deadline.

**Proof:** By contradiction. Suppose that a job \( \tau_i^h \in P_1^{\ell+} \) has not been assigned a deadline, and let \( \tau_i^\ell \) be the first job for which this happens at run-time.

Since \( d_i^h < d_i^\ell \) but \( \tau_i^h \) does not belong to \( P_1^{\ell+} \), it must exist a job \( \tau_s^s \in P_1^{\ell+} \) such that \( h < s < \ell \) and \( d_s^h < d_s^\ell < d_i^h \). Then, \( \tau_i^\ell \in P_1^{\ell+} \). Since \( \tau_s^s \) has been assigned a deadline at run-time, according to rule 3, \( \tau_i^\ell \) must have been assigned a deadline, against the hypothesis.

The following lemma proves that the absolute deadlines assigned by algorithm IDSP never exceed the absolute deadlines assigned by an algorithm that uses global time.

**Lemma 5:** Let \( a_i^\ell \) be the activation of the \( \ell \)-th instance of transaction \( T \). Under the assumption that the transaction is schedulable, the absolute deadline \( d_i^\ell \) of every job \( \tau_i^\ell \), computed dynamically using Equations (22)–(24), is never larger than \( a_i^\ell + D_1 \).

**Proof:** The complete proof has been removed for space constraints. We just report here the intuition behind it. The proof is by induction. First, we show that the lemma is true for the first job of the first instance. This is evident because rule 1 is the only one that can be applied. Then, we prove the induction step: if the lemma is true for all jobs in \( P_1^\ell \), then it is true for \( \tau_i^\ell \). This can be proved by showing how to compute \( d_i^\ell \), starting from rule 2 and rule 3.

**Lemma 5** guarantees that, if the transaction is locally schedulable on each node, then no task misses the deadlines that a global algorithm would have assigned on each node.

Now we want to prove that the protocol assigns deadlines so that the sporadic \( dbf \), as computed in Section V-B and V-A, is always respected.

**Lemma 6:** Let \( \tau_j^\ell \) be any job in \( P_1^\ell \). Under the assumption that the transaction is schedulable and all deadlines are assigned using Equations (22)–(24), the distance between \( d_j^h \) and \( d_i^\ell \) is never smaller than the distance as computed in Equation (21).

**Proof:** For \( \tau_j^\ell \in P_1^{\ell+} \), the lemma follows directly from Rule 3. For the other jobs, it is easy to see that we can apply a similar reasoning to the one in Lemma 4 to derive the thesis.

Now the main theorem.

**Theorem 1:** Under the assumption that all tasks execute for less than their WCET, and that for any interval the sum of the sporadic \( dbf \)s computed off-line never exceeds the length of the interval, then the \( dbf \) computed on-line by IDSP is always less or equal to the sporadic \( dbf \).

**Proof:** The proof has been removed for space constraints. We report here only the intuition behind it.
The proof is by contradiction. Let \([t_0, t_1]\) be the first and smaller interval in which the demand computed on-line exceeds the sporadic dbf. The, \(t_0\) must be coincident with a task activation and \(t_1\) with a deadline, let it be \(d_i\). According to Lemma 6, the deadlines in \([t_0, t_1]\) are separated by no less than their worst-case distance as computed by Equation (21). Then, we show that by moving deadline and activations in a conservative way (i.e. without decreasing the on-line demand), we reach one of the situations enumerated by the algorithms in Figures 7 and 6. Hence we obtain a contradiction, and the thesis is proved.

VII. CONCLUSIONS AND FUTURE WORK

In this paper we addressed the problem of analyzing the schedulability of sporadic transactions on a distributed system scheduled by EDF, and how to support our methodology at run time. We proposed an algorithm to compute the sporadic dbf offline on each node. We also proposed the IDSP protocol that assigns appropriate deadlines to jobs guaranteeing that the dbf computed on-line never exceeds the dbf computed off-line.

As a future work, we plan to extend the methodology to task graphs. Also, we would like to study the effect of combining different scheduling strategies on different nodes, and the effects of network scheduling.

REFERENCES


