Holistic Schedulability Analysis of a Fault-Tolerant Real-Time Distributed Run-time Support*

Pascal Chevochot Isabelle Puaut
IRISA, Campus de Beaulieu — 35042 Rennes — France (e-mail: pchevoch/puaut@irisa.fr)

Abstract

The feasibility test of a hard real-time system must not only take into account the temporal behavior of the application tasks but also the behavior of the run-time support in charge of executing applications. This paper is devoted to the schedulability analysis of a run-time support for distributed dependable hard real-time applications. In contrast to previous works that consider rather simple run-time supports (e.g. a real-time kernel made of a simple tick scheduler and an unreliable communication protocol), our work deals with a complex run-time support with fault-tolerance capabilities and made of multiple tasks that invoke each other.

1 Introduction

Essentially for economic reasons, a new trend in the design of dependable hard real-time systems is to rely on Commercial Off-The-Shelf (COTS) components (hardware and operating system). COTS hardware and operating systems are generally not designed for dependable applications. Using them in order to execute hard real-time applications with dependability requirements means that fault-tolerance mechanisms must be implemented in a middleware layer on top of the operating system. As a consequence, the overall complexity of the run-time support is increased compared to a run-time support without dependability requirements.

The feasibility test of a hard real-time system must not only take into account the temporal behavior of the application tasks, but also the behavior of the run-time support in charge of executing them. This task is simple when the run-time support is small (e.g. a real-time kernel made of a simple tick scheduler and an unreliable communication protocol), but becomes more complex when it includes elaborate functions such as fault-tolerance mechanisms.

This paper is devoted to the schedulability analysis of a run-time support for distributed dependable hard real-time applications, through an adaptation of the distributed system arbitrary deadline analysis developed by Tindell [8] for fixed priority scheduling. This run-time support, called HADES (*Highly Available Distributed Embedded System*) [1] was developed at the IRISA research center in Rennes, France, within the framework of a cooperation with the French department of defense (DGA) and Dassault-Aviation. Its objective is to execute applications in a predictable and reliable manner on top of COTS components (hardware and operating system).

Two approaches can be used to obtain task response times in distributed real-time systems. On one hand, one can simultaneously consider the application tasks and the run-time support tasks. On the other hand, the run-time support tasks can be considered separately; the knowledge of the system worst-case load scenario (e.g. worst-case arrival law of application tasks) is then used to compute the run-time support tasks response time. We have selected this second approach, because it allows to use a scheduling test for the application tasks different from the test for the run-time support tasks.

Single processor schedulability analysis for fixed priority tasks has received considerable attention. In the past ten years, the original fixed priority analysis [4] has been considerably extended, relaxing many of the assumptions of the original computation model (e.g. [3]). Single processor analysis for fixed priority has been also extended for network schedulability analysis (e.g. [7]) and for distributed system analysis (e.g. [8]). Several studies [8, 5, 2, 6] have considered the costs of a run-time support in the feasibility tests: [5, 2, 6] consider the influence of interrupts on the feasibility tests (static scheduling for the first reference, and dynamic priorities for the last two ones). The work of Tindell and Clark [8] deals with the schedulability analysis of a distributed system with tasks that communicate through message passing. In this work, the run-time support is rather simple: it is made of a tick scheduler and a simple unreliable TDMA communication procotol.

The contribution of this paper is to apply the arbitrary deadline analysis developed by Tindell [8] to compute the response times for the tasks of a run-time support for distributed dependable hard real-time applications. In contrast to previous works that consider rather simple systems, our work deals with a complex run-time support that incorpo-

*This work is partially supported by the French Department of Defense (DGA/DSP), #198C4980031303011.

Proc. of the 7th International Conference on Real-Time Computing Systems and Applications (RTCSA'00), Cheju Island, South Korea, december 2000.

rates fault-tolerance mechanisms, and which is made of multiple tasks that invoke each other.

The rest of the paper is organized as follows. We present the HADES run-time support and detail how it schedules and executes tasks in section 2. Then, we compute the response times of the run-time support tasks (section 3). We conclude in section 4. Appendix A gives an exhaustive list of the run-time support tasks.

2 Run-time support

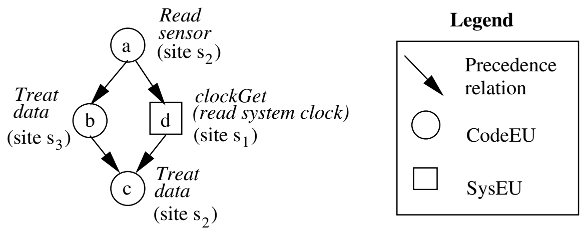
2.1 Architecture

Computations are distributed across several sites connected by a network. A site is made of a processor, a memory, a clock, and various interfaces (including a network interface, sensors, actuators). Resources (*e.g.* hardware resources, memory segments) can be shared between computations at a given site. Interrupts can come either from the clock or from the sensors.

2.2 Task model

Every task is modeled through a directed acyclic graph: a node, called an *EU* (*Elementary Unit*), models a computation local to a site; an edge models a *precedence relation* between two nodes. A precedence relation is called *local* (resp. *remote*) when its *EUs* are assigned to the same site (resp. different sites).

An *EU* can either be a *CodeEU* (*Code Elementary Unit*), *i.e.* a blocking-free computation, or a *SysEU* (*System Call Elementary Unit*), *i.e.* the request of a service of the run-time support (*e.g.* reading the system clock). Direct calls to the underlying real-time kernel are not allowed for the application tasks. An *EU* uses input variables and computes output variables. Two *EUs* from the same graph can be located on different sites (tasks are distributed). A simple example of distributed task is given below.



2.3 Overview of the HADES run-time support

The HADES run-time support is a middleware layer running on top of a COTS real-time kernel. The run-time support is in charge of providing a set of services which can be called by applications thanks to *SysEUs*. The run-time support is also in charge of controlling the tasks' execution: it is implicitly called at specific points during the execution of tasks, namely:

1. *Activation* of tasks, to achieve the operations required between a task *arrival* (time when the task is logically able to run) and its *release* (time when it is inserted in the run-queues) ;

2. *Validation* of precedence relations (a precedence relation from a to b is *validated* if the execution of a does not prevent b from executing anymore, *i.e.* if the execution of a is finished or the site of a is stopped) ;

3. *Termination* of task executions, to free the resources used by task executions.

In addition, the run-time support is executed concurrently with application tasks to handle interrupts, and to achieve treatments which are asynchronous with regard to application tasks (detection of sites stops or starts, clock synchronization, polling of network communication buffers).

The run-time support is decomposed into two software layers:

- A high level layer, named the *services layer*. Every service is modeled according to the task model shown in the previous paragraph. The service layer [1] comprises a *time service*, whose purpose is to launch periodic tasks, and a set of fault-tolerant distributed services (§ 2.3.1 to 2.3.4). Every service is made of several tasks. Some of them are entry points for the run-time support, and can be called thanks to *SysEUs*. For instance, the *SysEU* returning the current system time is a call to task `GetClock` of the time service.

  To avoid any confusion, in the following an application task is named *App_Task*, and a run-time support task is named a *RTS_Task*.

- A low level layer, named *dispatcher*. The dispatcher implements a fixed-priority FIFO scheduler for *RTS_Tasks*, while *App_Tasks* can have their own scheduling algorithm, which is implemented by a *scheduling service* made of a set of *RTS_Tasks* (*two-level scheduling*). The dispatcher maintains a priority-ordered set of run-queues containing the *EU*s of the tasks for which the execution is in progress. *EU* a has priority over *EU* b if a is in a run-queue with a higher priority than the one of b or if a is before b in the same run-queue. At any time, the eligible *EU* with the highest priority is executed. *App_Tasks* scheduled by a given scheduling algorithm are associated with a fixed range of priorities.

The dispatcher controls the execution of the non distributed tasks (their activation, the validation of their precedence relations, and the termination of their executions),

[1] For flexibility reasons, services have been designed so that each of them can be replaced by another service with the same interface but different properties [1]. Our run-time support comes with different versions of the multicast, failure detection and scheduling services. Only one service of each class is considered in the following.

while distributed tasks are managed by a *distributed execution control service* (*two-level execution control*). The tasks of the distributed execution control service are called by the dispatcher when a distributed task arrives and when the execution of one of its *EU* is finished.

The dispatcher also provides a small set of system calls. These calls are made thanks to *SysEU*s and are reserved for run-time support tasks, except for the `Invocation` system call which is used by application tasks to call services of the service layer. The system calls of the dispatcher are:

- `Invocation`: Call zero, one or several tasks (invocation mode = synchronous or asynchronous).
- `setClock`: Update the dispatcher clock.
- `ctrlRunQueue`: Family of system calls used to update the run-time support run-queues.
- `ctrlExec`: Family of system calls used to control task execution.

There are two major differences between *App_Tasks* and *RTS_Tasks*. First, the graph of *App_Tasks* is distributed (two *EUs* can be assigned to different sites), while the graph of *RTS_Tasks* is not. Second, *App_Tasks* have no direct access to the site hardware and to the real-time kernel, and are restricted to use the abstractions of the run-time support (resources, precedence relations, task invocation) to interact with one another.

A task (*RTS_Task* or *App_Task*) is released either due to an interrupt (e.g., interrupt generated by a sensor) or because it is called by another task. In the latter case, the call can be either *synchronous* (the caller is blocked) or *asynchronous*. There is no code executed in interrupt handlers; rather, an interrupt handler only releases the task in charge of handling the interrupt. At a higher level of abstraction, *App_Tasks* can be classified into two categories:

- *sensor-triggered App_Tasks*, whose arrivals are triggered by a sensor each time it triggers an interrupt to indicate that data can be read;
- the *time-triggered App_Tasks*, that are called periodically by a task of the time service, which is triggered by the clock interrupt.

We now detail the functionalities of the services which are required for the schedulability analysis of the run-time support. Details can be found in [1].

2.3.1 Failure detection and reliable multicast

The failure detection service is in charge of detecting site failures, allowing fail-free sites to determine on the set of failed and operational sites (sites are assumed to be *fail-silent*). It ensures the well-known properties of *real-time* Δ*-timeliness, vivacity* and *integrity*.

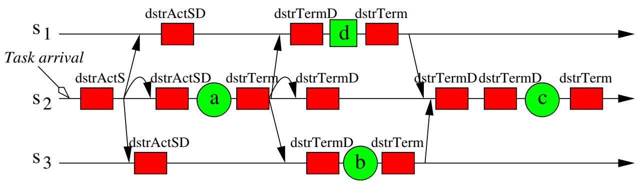
The multicast service is concerned with the (time-bounded) transmission of messages to a set of sites despite link omissions (we assume the existence of an upper bound on the number of successive network omissions, as well as an upper bound on the network latency). It ensures the well-known properties of Δ*-timeliness, validity, integrity agreement* and *virtual synchrony* and delivers messages according to *causal* ordering. To multicast a message m, a *RTS_Task* task i synchronously calls a task of the reliable multicast service running on the local site (named $mcastSend$), which simply inserts m into the output communication buffer, managed in FIFO ordering. A periodic task, named $mcastBg$, is used to send the messages of the output communication buffer and to call asynchronously the tasks in charge of handling the messages in the input communication buffer. $mcastBg$ also implements the failure detection service.

2.3.2 Clock synchronization

The clock synchronization service is concerned with keeping the sites clocks close from one another (the well-known properties of *precision* and *accuracy* are ensured). The protocol we have implemented is an adaptation of the algorithm of Lundelius and Lynch [9] to deal with a network with a bounded number of successive omissions instead of a reliable network as assumed in the original algorithm.

2.3.3 Distributed execution control

The distributed execution control service is made of a set of tasks, invoked when a distributed task is released or one of its *EUs* finishes. The above figure depicts the failure-free execution of a sensor-triggered *App_Task* (named i, which is the task taken in example in § 2.2) made up of four *EU*s located on sites s_1 to s_3.



In the figure, i is triggered by a sensor located on site s_2. A task of the distributed execution control service, named $dstrActS$ sends a message to *RTS_Tasks* $dstrActSD$ on sites s_1, s_2 and s_3, where the successors of *EU* a are located. On these sites, *RTS_Task* $dstrActSD$, which is a proxy for task $dstrActS$, uses a system call of the `ctrlExec` family to allocate the execution resources required for the execution of i. Task $dstrTerm$ is released at the end of the execution of each *EU* e; it sends a message to its proxy $dstrTermD$ on every site where a successor of e is located (on the example, *RTS_Task* $dstrTerm$ of site s_3 sends a message to site s_2). On a given site, *RTS_Task* $dstrTermD$ validates the precedence relations which result from an *EU* put on a remote site thanks to a system call of the `ctrlExec` family.

2.3.4 Scheduling

The scheduling service is concerned with the selection of the order in which the *EUs* of *App_Tasks* are to be inserted into the run-queues. Every scheduling service provides two tasks: task *schedAct* is released when a distributed task arrives; task *schedTerm* is released when an *EU* terminates its execution. The purpose of these two tasks is to update the dispatcher run-queues, using system calls of the `ctrl-RunQueue` family, according to a specific scheduling algorithm. One scheduling algorithm we have implemented is an extension to distributed tasks of the static planning-based scheduling algorithm of Xu and Parnas [10]. In this scheduling service, *schedAct* simply inserts *EUs* in the run-queues according to a schedule computed off-line, and *schedTerm* simply removes from the run-queues the *EU* that has just finished its execution.

2.4 Summary of run-time support characteristics

This paragraph summaries the assumptions and properties of the run-time support required for its schedulability analysis. Appendix A gives an exhaustive list of the run-time support tasks and interrupts. Two assumptions are made concerning the system architecture:

Assumption 1 *The time between two occurrences of an interrupt of level x is at least P_{int_x}.*

Assumption 2 *For any site s, the minimal time between a stop of s and its restart, and the minimum time between its start and its stop are bounded by a constant $\delta_{start/stop}$.*

Concerning the dispatcher, it has been designed to handle very basic functionalities, which makes its code small and thus easy to analyze. Thus, the following property holds:

Property 1 *The dispatcher is made of three local treatments with respective worst-case execution times: (i) $wcet_{term/start}$ to terminate an* EU *execution and to select the next* EU *to execute ($wcet_{term/start}$ integrates the cost of a context switch) ; (ii) $wcet_{sys(X)}$ to execute* SysEU X *(for the* `Invocation` SysEUs*, $wcet_{sys(\texttt{Invocation})}$ does not include the execution time of the called task) ; (iii) $wcet_{int}$ to take an interrupt into account and to release the task in charge of handling it.*

The following property comes from the implementation of the dispatcher, that must mask interrupts during the three above treatments in order to keep its internal data structures consistent:

Property 2 *The three treatments of the dispatcher (see property 1) cannot be preempted.*

A set of properties hold for the run-time support tasks (pr_i=priority of task i):

Property 3 *If two* RTS_Task *i and j use the same resource, then $pr_i = pr_j$.*

Property 4 *Let i be a* RTS_Task*. For all* RTS_Task *j called synchronously by i, $pr_j < pr_i$ (the priority of j is higher than the one of i).*

Property 5 *Every* CodeEU *e of a* RTS_Task *has a known worst-case execution time $wcet(e)$.*

Property 6 *There is no cycles in the call graph of* RTS_Tasks.

Property 7 *The reliable multicast service transmits data from the output buffer of every site every Δ_{send}. Moreover, the time from the retrieval of a message from the head of the emission buffer on a site, and the call of the task in charge of handling it on one of the target sites is less than Δ_{tr}.*

Property 8 *Local clocks are synchronized with a precision of γ.*

Property 9 *Every site detects the stop or the start of another one in at most δ_{detect}.*

For details, the expression of Δ_{send}, Δ_{tr} and Δ_{detect} is given in [1]. The expression of γ is given in [9].

Finally, by construction, *App_Tasks* verify two properties:

Property 10 *If i is an* App_Task *and j in an* RTS_Task*, then i is inserted into a run-queue having a lower priority than the one where j is inserted.*

Property 11 App_Task*s do not use asynchronous* `Invocation` SysEUs.

3 Schedulability analysis

This paragraph is devoted to a description of the schedulability analysis of the distributed run-time support presented in section 2. The analysis relies on the results of the work of Tindell [8] about fixed priority schedulability analysis (see § 3.1 for a summary). The remainder of this section is then devoted to a detailed description of the analysis. For the analysis, we decompose *RTS_Tasks* in two subsets: $Sync$ is the set of *RTS_Tasks* which are called synchronously and $ASync$ the remaining *RTS_Tasks*, that execute in the background. Let $sync_i$ be the set of tasks synchronously called by a task i.

The analysis is done in two steps. First, we analyze the response time r_i of every task i in $ASync$ on a per-site basis (§ 3.2); r_i includes the cost of tasks called synchronously by i (i.e. tasks in $sync_i$). With the knowledge of r_i (i in $ASync$), we compute for every *EU* e its worst-case execution time, costs of $ASync$ *RTS_Tasks* included (named $\overline{C}_e$).

Secondly, we perform the analysis on a system-wide basis (§ 3.3); the worst-case communication latency is computed and is used to obtain the worst-case time required to activate a distributed task i (named δ_{actT_i} for a time-triggered *App_Task* and δ_{actS_i} for a sensor-triggered *App_Task*) and the worst-case time required to validate a distributed precedence relation $e \rightarrow e'$ (named $\delta_{e \rightarrow e'}$).

3.1 Tindell's results

A task is an infinite sequence of executions of a sequential program. Every task i is assigned a priority pr_i. We note $hp(i)$ the set of tasks of higher priority than task i assigned to the same processor. Priorities are assumed to be unique, and at any time the task with the highest priority executes. For each arrival, a task i may execute a bounded amount of computation C_i. Each task is *periodic* (resp. *sporadic*), *i.e.* there is an exact (resp. a minimal) delay P_i between successive arrivals of task i. Each task i may lock/unlock semaphores according to the priority ceiling protocol or equivalent so that the longest time that task i could wait spend blocked by a lower priority task is bounded by a constant B_i.

There may be a delay between a task *arrival* (time when a task is logically able to run) and its *release* (time when it is inserted in the run-queues); this can occur for instance if the scheduler uses polling to fill in the run-queue. This delay is named the *release jitter* and is noted J_i.

The worst-case response time of task i, termed r_i, is the worst-case time between the arrival of task i and its completion. Assuming that for all j, P_j, B_j, C_j and J_j are known, r_i can be computed as follows:

$$r_i = J_i + \underset{q=0,1,2,\ldots}{Max} \{w_i(q) - q.P_i\} \qquad (1)$$

$$w_i(q) = (q+1)C_i + B_i + \sum_{\forall j \in hp(i)} \lceil \frac{J_j + w_i(q)}{P_j} \rceil .C_j \qquad (2)$$

$w_i(q)$ is the length of an i-busy period which is initiated by a release of task i and which contains $q+1$ executions of i. A i-busy period is the time when the processor is continuously occupied by requests of priority $pr(i)$ or higher. In formula 2, $\lceil \frac{J_j + w_i(q)}{P_j} \rceil$ is the number of releases of task j during $w_i(q)$. In formula 1, $w_i(q) - qP_i$ is the time between the $q+1$th release of task i and its completion. The above equations produce a result if the processor utilization is less than 100% [8].

When tasks do not have distinct priorities, applying formula 2 would lead to an overestimation of r_i, because the formula considers that every task j can preempt task i during the $q+1$th execution of task i. We thus propose to replace it by the following equation:

$$w_i(q) = (q+1)C_i + B_i + \sum_{\forall j \in hp(i)} \lceil \frac{J_j + w_i(q)}{P_j} \rceil .C_j + \sum_{\forall j \in sp(i)} (1 + \lfloor \frac{J_j + q.P_i}{P_j} \rfloor).C_j \qquad (3)$$

with $sp(i)$ the tasks having the same priority than task i. The formula assumes that tasks with the same priority are managed in FIFO order, and that there is no tie breaking rule if two tasks arrive simultaneously. For space consideration, the proof of this formula is left to the reader.

3.2 Single processor schedulability analysis

In this paragraph, we compute for every *EU* e its worst-case execution time, run-time support costs included ($\overline{C}_e$). We first compute the worst-case response time r_i of tasks i in $ASync$ (r_i includes the worst-case response time of all the tasks called in $sync_i$), and then compute $\overline{C}_e$.

3.2.1 Response time of a *RTS_Task* in $ASync$

Equations 1 and 3 can be used in a straightforward manner to compute the response time of *RTS_Tasks* on every site, as far as the values of P_i, C_i, B_i and J_i are known for every task i:

$$\underset{i \in ASync}{r_i} = J_i + \underset{q=0,1,2,\ldots}{Max} \{w_i(q) - q.P_i\} \qquad (4)$$

$$w_i(q) = (q+1)C_i + B_i + \sum_{\forall j \in ASync\ st\ j \in hp(i)} \lceil \frac{J_j + w_i(q)}{P_j} \rceil .C_j + \sum_{\forall j \in ASync\ st\ j \in sp(i)} (1 + \lfloor \frac{J_j + q.P_i}{P_j} \rfloor).C_j + \sum_{1 \le x \le Nb_{int}} \lceil \frac{w_i(q)}{P_{int_x}} \rceil .wcet_{int} \qquad (5)$$

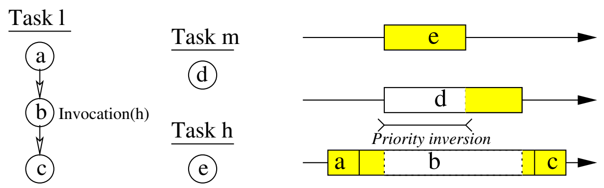
Equation 4 is identical to equation 1. The only difference between equation 5 and the original one (equation 3) is the addition of the last term which adds the execution costs of the interrupt handlers, considered as pseudo-tasks with the highest priority and with worst-case execution time $wcet_{int}$. Nb_{int} is the number of interrupt levels. The minimum delay P_{int_x} between two interrupts of level x is known, thanks to assumption 1.

Computation of C_i: Thanks to properties 1 and 5, the worst-case execution time of any *EU* of a *RTS_Task* $i \in ASync$ is known, which allows to compute C_i for every task i in $Async$:

$$C_i = \sum_{e \in eu_i} \begin{cases} wcet(e) + wcet_{term/start} & \text{if } e \text{ is a } \textit{CodeEU} \\ wcet_{sys(e)} & \text{if } e \text{ is a } \textit{SysEU} \end{cases} + \sum_{j \in sync_i} C_j \qquad (6)$$

The first sum corresponds to the worst-case execution time of the *EU*s of *RTS_Task* i. The second one corresponds to the worst-case execution time of tasks synchronously called by i. Due to the absence of cycle in the call graph of *RTS_Tasks* (property 6), the recursion on the computation of C_i is bounded.

Computation of B_i: Thanks to property 3 and 4, the use of the resources can not lead to multiple priority inversions. However, priority inversion can occur because of synchronous invocations, as depicted below. In the figure, task l, with lower priority, calls task h with higher priority, which causes task m with an intermediate priority to be blocked.



Such a priority inversion is taken into account as follows:

$$B_m = Max\Big(\bigcup_{\substack{l,h\ st\ h\in RTS\_Tasks\ \wedge \\ h\in hp(m)\ \wedge\ m\in hp(l)\wedge\ h\in sync_l}} C_h \ \cup \{wcet_{term/start}\} \cup \{wcet_{sys}\}\Big) \qquad (7)$$

The first term corresponds to the worst-case execution times of *RTS_Tasks* which have higher priorities than task m and which are synchronously called by tasks having lower priorities than task m. The lower priority tasks can encompass both *RTS_Tasks* and *App_Tasks*. The second and last terms correspond to dispatcher treatments that are triggered by lower priority tasks and that are by construction (property 2) non preemptable.

Computation of P_i: By construction (assumption 1), the time interval between two interrupts of level x is at least P_{int_x}. Thus, the pseudo-period P_i of a task i triggered by an interrupt of level x is P_{int_x}. In our run-time support, there is only two interrupt-triggered *RTS_Tasks*, named *clockWakeup* and *dstrActS*. So, the pseudo-period of task $clockWakeup$ is P_{int_0} (0 is the interrupt level used by the hardware clock). We come back on task $dstrActS$ in the following (§ 3.3.2).

For a *RTS_Task* i asynchronously called by another *RTS_Task* j, a pessimistic value for P_i is the one of P_j. However, due to the interface of the `Invocation` system call, a task i including an `Invocation` system call of task j does not necessarily call j every time it is executed. For instance, task $clockWakeUp$ of the time service calls a periodic task k only once every x, with x the period of k in multiples of clock ticks (i.e. P_{int_0}). In this situation, as well as in every other situation where a task i is called by a task j only once every x executions of j, then $P_i = x.P_j$. This allows to compute P_i for all periodic tasks of the run-time support (tasks $dstrActT, csyncCorr, mcastBcastClock$ and $csyncEstim$, see appendix A). For *RTS_Tasks* asynchronously called by $mcastBg$, assumption 2 allows to replace task $dstrNC$ (see appendix A for its role), whose arrival law is hard to predict, by $n-1$ sporadic pseudo tasks having the same priority than $dstrNC$, with n the number of sites. Their pseudo-period is $\delta_{start/stop}$. Thus $x = \delta_{start/stop}/P_{mcastBg}$. We deal with $dstrTermD$ and $dstrActSD$ in the following (§ 3.3.2).

Computation of J_i: For an interrupt-triggered *RTS_Task* i, since the worst-case time for the dispatcher to take an interrupt into account and launch i is $wcet_{int}$, then $J_i = wcet_{int}$. For a *RTS_Task* i asynchronously called by a *single RTS_Task* j, its release jitter J_i is the worst-case response time of j (*i.e.* r_j).

3.2.2 Worst-case execution time of an *EU* ($\overline{C}_e$)

In this paragraph we compute the worst-case execution time ($\overline{C}_e$) of an *EU* e of an *App_Task* including the run-time support execution costs. Thanks to property 10, *App_Tasks EUs* have lower priorities than *RTS_Tasks EUs*. So, $\overline{C}_e$ can be computed using equation 8:

$$\overline{C}_e = \begin{cases} C_e + wcet_{term/start} & \text{if } e \text{ is a } \textit{CodeEU} \\ wcet_{sys} + C_i & \text{if } e \text{ sync. calls } i \end{cases} + C_{dstrTerm}$$
$$+ \sum_{\forall j \in ASync} \lceil \frac{J_j + \overline{C}_e}{P_j} \rceil .C_j + \sum_{1\le x \le Nb_{int}} \lceil \frac{\overline{C}_e}{P_{int_x}} \rceil .wcet_{int} \qquad (8)$$

The first term is the WCET of e, run-time support costs excluded. In the case e is an `Invocation` *SysEU*, this term takes into account the cost of the called task. The second and third terms are the WCETs of tasks $dstrTerm$ (see § 2.3.3) and $schedTerm$ (see § 2.3.4), which are implicitly called in synchronous mode when the execution of e is finished. Finally, the last two terms account for the preemptions of e by higher priority activities.

3.3 System-wide schedulability analysis

We first apply our extension of Tindell results to compute the worst-case message transmission latency, which allows us to compute the worst-case time required to activate a distributed task and validate a remote precedence relation. In the following, to avoid any misunderstanding, every task i running on a site s is noted $i_{/s}$.

3.3.1 Worst-case message transmission latency

To compute the worst-case time, noted b_m, that a message m can spend in the emission buffer of a site, it is necessary to know the pseudo-period P_m at which a message m is requested to be sent, and the message release jitter J_m, i.e. the worst-case delay between the arrival of the sender task and its insertion into the output communication buffer: $P_m = P_{sender_m}$ and $J_m = r_{sender_m}$, with *sender* the sender task. In the figure of appendix A, we can see that $sender_m$ is task $dstrTerm$ or $dstrActS$. We give the expressions of P and J for these tasks in § 3.3.2,.

To compute b_m, equations 1 and 3 can be used (see equations 9 and 10 below). In equation 10, $hp(m) = \emptyset$ is empty because the communication buffer is managed in FIFO ordering (all messages have the same priority). In equation 10, Δ_{send} is the time between two transmissions of messages of the buffer (see property 7).

$$b_m = \underset{q=0,1,2,\ldots}{Max} \{w_m(q) - q.P_m\} \tag{9}$$

$$w_m(q) = \left((q+1) + \sum_{\substack{\forall n \neq m \ st \\ s_n = s_m}} (1 + \lfloor \frac{J_n + q.P_m}{P_n} \rfloor) \right) .\Delta_{send} \tag{10}$$

Knowing b_m, it is then possible to compute a_m which is the worst-case time between the arrival of a task that sends a message m and the arrival of the task in charge of treating the message on the target site: $a_m = b_m + \Delta_{tr}$. In this equation, Δ_{tr} is the worst-case time required to transfer a message on the communication medium (property 7). Time Δ_{tr}, like b_m, depends on the algorithm used for multicast service and on r_{commBg}.

3.3.2 Worst-case time to activate a distributed task

This paragraph is devoted to the computation of the worst-case time to activate a distributed task, i.e. the worst-case time between the task arrival and its insertion into the run-queues of the sites onto which it executes. We assume that the list of distributed *App_Tasks* (sensor and time-triggered) is known, as well as the tasks arrival law.

Sensor-triggered *App_Tasks*: Recall that all sensor-triggered *App_Tasks* of a given site s are called by a single task $dstrActS_{/s}$ (see § 2.3.3 and appendix A). Because different sensors can request interrupts at different rates, the arrival law of task $dstrActS_{/s}$ is neither periodic nor sporadic. Therefore, for the sake of schedulability analysis, we replace this task by a set of sporadic tasks $dstrActS_{i/s}$, with i a sensor-triggered *App_Task*. The pseudo-period of $dstrActS_{i/s}$ is P_{int_x}, with x the interrupt level which triggers i, and its worst-case release jitter is $wcet_{int}$.

Similarly, on every site, task $dstrActSD_{/s}$ is replaced by a set of pseudo-tasks $dstrActSD_{i/s}$, one for each sensor-triggered *App_Task*. A message m_i is sent from $dstrActS_{i/s}$ to $dstrActSD_{i/r}$. $dstrActSD_{i/s}$ has for pseudo-period the one of $dstrActS_{i/s}$, and for worst-case release jitter $r_{dstrActS_{i/s}} + a_{m_i}$. Thus, the worst-case time to activate a sensor-triggered *App_Task* i, noted δ_{actS_i} is $\delta_{actS_i} = \underset{\forall s \in S_i}{Max}\{r_{dstrActDS_i/s}\}$, with S_i the set of sites where i executes.

Time-triggered *App_Tasks*: On every site s, task $dstrActT_{/s}$ is in charge of calling time-triggered distributed *App_Tasks*. As it manages *all* time-triggered distributed *App_Tasks*, $dstrActT_{/s}$ is neither periodic nor sporadic. For the sake of schedulability analysis, we replace on every site task $dstrActT_{/s}$ by a set of sporadic tasks $dstrActT_{i/s}$, one of each time-triggered distributed *App_Task* i. Task $dstrActT_{i/s}$ has the same period as i and has $r_{clockWakeUp_{/s}}$ as release jitter. The worst-case time to activate a time-triggered *App_Task* i, noted $\delta_{actT_{i/s}}$, is then $\delta_{actT_i} = \gamma + \underset{\forall s \in S_i}{Max}\{r_{dstrActT_i/s}\}$, with γ the precision of clock synchronization (property 8). γ depends on the protocol used for clock synchronization service and on $r_{mcastBcastClock}$, $r_{csyncEstim}$, $r_{csyncCorr}$ and r_{commBg}.

3.3.3 Worst-case time to validate a remote precedence relation

Recall that each time an *EU* e terminates on a site s, *RTS_Task* $dstrTerm_{/s}$ is called. $dstrTerm_{/s}$ multicasts a messages to tasks $dstrTermD_{/r}$ of every site r hosting an *EU* e' such that $e \rightarrow e'$. Since there is a single task $dstrTermD_{/s}$ per site to validate all precedence constraints, its arrival law is neither periodic nor sporadic. Therefore, as an artifice, task $dstrTermD_{/s}$ is replaced by a set of pseudo-tasks $dstrTermD_{e \rightarrow e'/s}$ for all precedence relations $e \rightarrow e'$ with e' located on site s. The pseudo-period of $dstrTermD_{e \rightarrow e'/s}$ is the one of e and its worst-case release jitter is $r_{e/s} + r_{m_e}$ with m_e the message use by $dstrTerm_{/s}$ to validate this precedence relation.

Two cases have to be considered to compute the worst-case time required to validate a precedence relation $e \rightarrow e'$:

- The site s of e is not stopped. In this situation, we have $\delta^{ns}_{e \rightarrow e'} = r_{DstrTermD_e/s}$.
- The site of e stops during the validation of the precedence relation. Since site stops are detected in at most δ_{detect} (property 9), we have $\delta^{s}_{e \rightarrow e'} = \delta_{detect} + r_{dstrNC/s_{e'}}$

Thus, the worst-case time to validate the precedence relation from e to e' is the maximum of the two above values $\delta_{e \rightarrow e'} = Max(\delta^{ns}_{e \rightarrow e'}, \delta^{s}_{e \rightarrow e'})$.

3.3.4 Holistic analysis

The scheduling equations (equations 5, 8 and 10) are mutually dependent. For example, for *RTS_Task* $i_{/s}$, $J_{i_{/s}}$ can depend on b_m, with m sent by *RTS_Task* $j_{/s'}$. So, b_m depends on $r_{j_{/s'}}$, which can depend on *RTS_Task* $k_{/s}$ which has a higher priority than $i_{/s}$. Nevertheless, the equations used to compute r_i have a solution, as shown by Tindell in [8], because all the used scheduling equations are monotonic in window size w, response time, and release jitter. To find

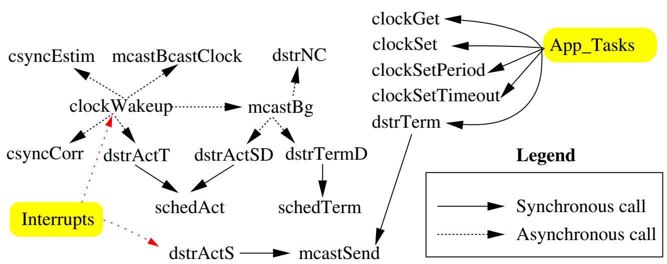
Name	Prio	Arrival	Description
clockWakeup	0	Interrupt	Increments the system clock & wakes-up periodic tasks
clockGet	0	Call (sync)	Reads the system clock
clockSet	0	Call (sync)	Updates the system clock
clockSetPeriod	0	Call (sync)	Registers a task so that it is released periodically
clockSetTimeout	0	Call (sync)	Requests the release of a task
mcastSend	1	Call (sync)	Inserts a message into the communication buffer
mcastBcastClock	1	Call (async)	Inserts a timestamped message into the output communication buffer for the purpose of clock synchronization
mcastBg	1	Call (async)	Accesses periodically the network hardware to send the messages of the output communication buffer and to deliver the messages of the input communication buffer
csyncEstim	2	Call (async)	Estimates and stores the value of a remote clock
csyncCorr	2	Call (async)	Corrects the local clock
schedAct	3	Call (sync)	Inserts the *EUs* of an *App_Task* into the run-queues
schedTerm	3	Call (sync)	Deletes a terminated *EU* from the run-queues
dstrActS	4	Interrupt	Manages the arrival of a sensor-triggered *App_Task*
dstrActT	4	Call (async)	Manages the arrival of a time-triggered *App_Task*
dstrTerm	4	Call (sync)	Manages the termination of an *App_Task EU*
dstrActSD	4	Call (async)	Allocates execution structures for an *App_Task* execution on a given site
dstrTermD	4	Call (async)	Validation of a precedence relation
dstrNC	4	Call (async)	Notification of a site stop or start
idle	255	Boot	Idle task (infinite loop)

the solution, we adopt the solution proposed by Tindell: all the release jitters are considered null, which permits to compute a first estimation of response times and then of release jitters, and so on, until each response time converges, which is guaranteed if the system is not overloaded.

4 Conclusion

In this paper, we have computed the response times of a complex run-time support tasks using a holistic approach, which is an adaptation of the distributed system arbitrary deadline analysis developed by Tindell for fixed priority scheduling. To be able to use Tindell's analysis, most tasks of the run-time support have been designed as periodic or sporadic tasks. For example, network management is achieved through periodic polling of the network buffer rather than using interrupts. For the remaining tasks (*e.g.* the task executed when a distributed task arrives), an artifice (transformation of tasks with an unknown arrival law into n sporadic tasks) has been used. Several extensions of the original analysis have been required to analyze our run-time support: extension to tasks that invoke other ones and extension to tasks with identical priorities, which were not supported in the original analysis.

A List of run-time support tasks



The table above describes every task of the service layer. It gives for every task its name, its priority, its arrival mode and a brief description of its role. The figure below gives the run-time support call graph.

References

[1] E. Anceaume, G. Cabillic, P. Chevochot, and I. Puaut. A flexible run-time support for distributed dependable hard real-time applications. In *ISORC*, pages 310–319, May 1999.

[2] K. Jeffay and D. Stone. Accounting for interrupt handling costs in dynamic priority task systems. In *RTSS*, pages 212–221, Dec. 1993.

[3] J. Lehoczky. Fixed priority scheduling of periodic task sets with arbitrary deadlines. In *RTSS*, pages 201–209, Dec. 1990.

[4] C. Liu and J. Layland. Scheduling algorithms for multiprogramming in a hard real-time environment. *Journal of the ACM*, 20(1):46–61, Jan. 1973.

[5] K. Sandström, C. Eriksson, and G. Fohler. Handling interrupts with static scheduling in an automotive vehicle control system. In *RTCSA*, Oct. 1998.

[6] M. Spuri. Analysis of deadline scheduled real-time systems. Research report RR-2772, INRIA, France, Jan. 1996.

[7] J. Strosnider, T. Marchok, and J. Lehoczky. Advanced real-time scheduling using the IEEE 802.5 token ring. In *RTSS*, pages 42–52, 1988.

[8] K. Tindell and J. Clark. Holistic schedulability analysis for distributed hard real-time systems. *Microprocessors and Microprogramming*, 40:117–134, 1994.

[9] J. L. Welch and N. Lynch. A new fault-tolerant algorithm for clock synchronization. *Information and Computation*, 77:1–36, 1988.

[10] J. Xu and D. Parnas. Scheduling processes with release times, deadlines, precedence, and exclusion relations. *IEEE Trans. on Software Engineering*, 16(3):360–369, Mar. 1990.

Acknowledgements We thank Pascal Richard from LISI for his comments and helpful suggestions on this work.