Performance analysis of various scheduling algorithms for real-time systems composed of aperiodic and periodic tasks

J. GOOSSENS  C. MACQ
Département d’Informatique, Université Libre de Bruxelles
Campus Plaine, CP 212
Boulevard du Triomphe, Brussels, 1050, Belgium

Abstract
We consider a simulation tool for the study of scheduling algorithms for real-time systems composed of periodic and aperiodic tasks. We first introduce the scheduling problem of real-time systems, we present popular scheduling algorithms for periodic tasks with hard deadlines, and then we consider the case of mix systems, composed of periodic as well aperiodic tasks and we briefly present the various popular scheduling algorithms for these kind of systems. After that, we describe our simulation tool and define the metrics we use to compare the performances of the various servers. In a next section, we present results of simulations and classify the aperiodic servers. In this section, we also present strategies to improve the performances of the various scheduling algorithms. Finally, we give some concluding remarks.

Keywords: real-time systems, scheduling, rate monotonic, deadline monotonic, deadline driven scheduler, background processing, polling server, deferrable server, priority exchange server, sporadic server, total bandwidth, slack stealing, response time, tardiness, number of preemptions, first in first out, earliest deadline first, least laxity first.

1. Introduction
We shall consider a simulation tool for the simulation and the study of scheduling algorithms for real-time systems composed of aperiodic and periodic tasks.

Real-time systems are characterized by stringent timing constraints; hence, the correctness of a task computation depends not only on its logical or computational results, but also on the instant when the result is made available. The most important feature of a real-time systems is its predictability [10], i.e., the ability to determine whether the system is capable (or not) to meet all the timing requirements of the tasks. Examples of such systems include the control of engines, traffic, nuclear power plants, time-critical packet communications, aircraft avionics and robotics. Typically, the timing constraint of a task is a deadline; we can distinguish between two kinds of deadline. If meeting a task deadline is critical for the system functionality, then the deadline is said to be hard; missing a hard deadline is considered a definite failure, and leads to catastrophic consequences. If it is desirable to meet a task deadline, but occasionally missing it can be tolerated, then the deadline is said to be soft; a task with a soft deadline is expected to be completed either before the deadline or as early as possible after it. The deadlines of soft tasks are often not specified, meaning that they are identical to the computation time.

There are two kinds of real-time tasks, depending on their arrival pattern: periodic tasks (the task has a regular inter-arrival time called the period, a deadline and a computation time) and aperiodic tasks (the task can arrive at any time; such a task is characterized by a computation time and a deadline; the latter is usually considered as soft).

We consider real-time systems constituted of a set of $n$ periodic tasks: $\tau_1, \ldots, \tau_n$ and a set of aperiodic tasks: $J_1, J_2, \ldots$. Each periodic task (say $\tau_i$, $1 \leq i \leq n$) is characterized by the quadruple $(T_i, D_i, C_i, O_i)$ with $0 < C_i \leq D_i$, $C_i \leq T_i$ and $O_i \geq 0$, i.e., by a period $T_i$, a hard deadline $D_i$, an execution time $C_i$ which can be considered as the worst-case execution time for a request of $\tau_i$, and an offset $O_i$ giving the instant of the first request. The execution of the $k$th request of task $\tau_i$, which occurs at time $O_i + (k-1)T_i$, must finish before or at time $O_i + (k-1)T_i + D_i$. We define the periodic load $U_p = \sum_{i=1}^{n} \frac{C_i}{T_i}$ as the (long term) fraction of processor time spent in the execution of the periodic tasks (if schedulable), we shall later see the interest of this notion. Each aperiodic task (say $J_k$) is characterized by the tuple $(a_k, p_k, d_k)$ where $a_k$ is the arrival time of the task, $p_k$ its execution time and $d_k$ its (soft) deadline, contrary to what happens for periodic tasks, those characteristics are not known beforehand.

Concerning the scheduling algorithm we shall first consider the problem of scheduling only periodic tasks and then we shall extend the algorithms in order to schedule mixed systems composed of periodic and aperiodic tasks as well.

2. Scheduling of periodic tasks
Among the various scheduling algorithms proposed for periodic tasks, we can distinguish between static and
For static schedulers, the scheduling algorithm determines the priorities of the tasks beforehand, based on the task characteristics (e.g., the worst case execution time, the deadline, the period, etc.). During the execution of the system, the scheduler selects the highest priority request, i.e., the active request which corresponds to the highest priority task (if there are many active requests corresponding to this task which may happen if \( D_i > T_i \), generally the oldest one is selected). We consider in this paper preemptive algorithms: a running request may be interrupted at any time to give the CPU to a request with a higher priority. The most popular static schedulers are the rate and the deadline monotonic schedulers which assign the priorities to each task in inverse proportion of the periods and the deadlines, respectively [4].

For dynamic schedulers, the scheduling algorithm computes the priorities of the various requests during the execution of the system. The priority of each active request is based on the system state, e.g., the current time, the request characteristics (e.g., the remaining execution time of each request, the time available before reaching the deadline), etc. It follows that the priority of a task or a request may change during system evolution. The most popular dynamic (and optimal) schedulers are the **deadline-driven** and the **least laxity first** schedulers. The deadline-driven scheduler (also termed the **earliest deadline scheduler**, or **earliest deadline first**) is a dynamic scheduling algorithm which gives (at any instant) the highest priority (and then the CPU) to the active request with the nearest deadline. The least laxity first scheduler [6] (also termed **slack-time algorithm**) is based on the **laxity** of the requests, which is defined as the maximal amount of time that the request can wait while still meeting its deadline. The least laxity first scheduler gives (at any instant) the highest priority (and then the CPU) to the active request with the smallest laxity. Ties, if any, are generally broken arbitrarily.

### 3. Scheduling of mixed systems

We shall consider here systems composed of periodic tasks with hard deadlines and aperiodic tasks with soft deadlines. The scheduling algorithm must in this case satisfy all hard deadlines of the periodic tasks while minimizing the response times of aperiodic tasks. We can distinguish between two kinds of scheduling algorithms: **bandwidth-preserving servers**, which use an additional periodic task (or several ones) in order to “serve” the aperiodic requests, and **slack stealing algorithms** which are based on the computation of the **slack time**, i.e., the time available to aperiodic requests without affecting the schedulability of periodic requests.

### Bandwidth-preserving servers

The server is in this case an additional periodic task which serves the aperiodic requests. Regarding the priorities we apply a “classical” (static or dynamic) priority-driven scheduler. The various bandwidth-preserving servers differ in the strategy chosen to handle the capacity of the server, i.e., the (remaining) processing time that the server can use to serve aperiodic tasks.

A first (trivial) scheduler in this category is the **background servicing** [3], which serves the aperiodic tasks when no periodic task is active, in other words when the system composed of the periodic tasks only is idle; this amounts to introduce a server with infinite period and capacity, but with the lowest possible priority.

A basic strategy with a truly periodic server is the **polling server** [3]: when the server becomes running (i.e., has the highest priority) the capacity of the server decreases and is used (while the capacity is positive) to serve aperiodic request(s), if any otherwise the capacity of the server becomes 0 and will be restored with the next request of the server. The major drawback of the polling server is that the capacity of the server is lost whenever no aperiodic request exist when it starts running, even if, a new aperiodic request occurs slightly later. A simple improvement consists to decrease (and not reset) the capacity in the situation described above; this leads to the **extended polling server** [5]. The **deferrable server** and the **priority exchange server** [3] overcome also the drawback of the polling server; the capacity is in this case used for lower priority (periodic) requests or new aperiodic requests. We shall not give more details here.

In the previous algorithms the capacity of the server is restored periodically; the **sporadic server** is quite different regarding the restoration policy, which is not periodic and depends on its previous utilization; details of the algorithm used to determine the restoration instants and restoration amounts can be found in [8].

Let us finally mention the total bandwidth server [9] which is defined only for the deadline-driven scheduler and consists in assigning a well-chosen deadline to each aperiodic task and applying for the whole system the deadline-driven rule. This is not exactly a bandwidth-preserving server, but is quite akin them.

### Slack stealing

This approach does not create a periodic server for aperiodic task servicing. Rather it creates a passive task, the **slack stealer**, which steals some processing time of periodic tasks (and consequently delays the periodic requests) for aperiodic tasks without causing hard deadline failures. The algorithm is divided in two phases: the computation of the slack time (the
time which can be stolen to the periodic request) and the slack stealer itself. This approach minimizes the response time of aperiodic tasks while preserving the schedulability of periodic tasks. However the time and the space complexities of this approach are generally proportional to the least common multiple of the periods of all periodic tasks, so that this approach is not adapted to most real-time systems.

The Slack stealing scheduling was first defined for the static rate monotonic scheduler [2] and then adapted to the dynamic deadline driven scheduler [7].

Table 1 summarizes the time and the space complexities of the various algorithms presented in this section, where \( m \) is the maximal number of simultaneous aperiodic requests and \( T_r \) the period of the server. Table 1 shows also if the scheduling algorithm is compliant (\([\) or not \( ]\)) with a static or a dynamic priority based rule. When we implemented of the various algorithms from their “formal” definitions in the literature we had to lift some imprecisions and ambiguities; Table 1 displays also this aspect (see [5] for details).

### Table 1: Complexities, implementation and static/dynamic compliance of the various algorithms.

<table>
<thead>
<tr>
<th>Server</th>
<th>Time complexity</th>
<th>Space complexity</th>
<th>Implementation work</th>
<th>Static</th>
<th>Dynamic</th>
</tr>
</thead>
<tbody>
<tr>
<td>Background</td>
<td>( O(1) )</td>
<td>( O(1) )</td>
<td>obvious</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Polling</td>
<td>( O(1) )</td>
<td>( O(1) )</td>
<td>simple</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Deferrable</td>
<td>( O(1) )</td>
<td>( O(1) )</td>
<td>simple</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Priority excl.</td>
<td>( O(n) )</td>
<td>( O(n) )</td>
<td>hard</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Sporadic</td>
<td>( O(1) )</td>
<td>( O(\frac{1}{T_r}) )</td>
<td>simple</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Total band.</td>
<td>( O(m \cdot n) )</td>
<td>( O(1) )</td>
<td>simple</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

4. Simulation Tool and Metrics

In this section, we briefly describe the tool we developed to analyze the various aperiodic servers, but also the metrics we used to compare the performances of the different algorithms. This tool has three important objectives: portability, flexibility and easy use.

Portability is an important goal; it is here obtained through the usage of the Java programming language to implement the simulator.

Flexibility is very useful when we want to add new scheduling algorithms. To reach this goal, an object-oriented approach [1] has been used for the analyze and the design of the simulator. This approach leads to minimize the dependencies between the different objects, with a very interesting consequence: the addition of a new server induces very few classes must augmented or added.

Finally, easy of use is a guarantee that this tool can be used by other people than the developers. This was (hopefully) obtained with the deployment of a user-friendly graphical interface.

We shall now consider the different metrics used in this paper to study the performances of the various algorithms. The first one, and certainly the most popular one, is the response time of aperiodic requests: if an aperiodic request \( J_i \) becomes active (ready to run) at time \( c_i \) and ends its computation at time \( f_i \), the response time is equal to the difference \( f_i - c_i \).

A second metric, very interesting in our model (where aperiodic requests are also characterized by a soft deadline), is the tardiness (or exceeding time) of aperiodic requests, which is equal to \( \max(0, f_i - c_i - d_i) \). This metric is more original and is used to obtain another view of algorithm performances.

As said in the introduction, predictability is the most important property in a real-time system. In this respect, the number of preemptions for periodic or aperiodic tasks may also be interesting to compare different servers. In fact, preemptions may be very costly operations in actual computers and too a large number of those implies additional delays, which may be non-neglectible, contrary to what was assumed.

5. Experimental results

All the algorithms described in section 3 have been simulated to compare their respective performances, with the metrics defined above. Simulations on dynamic servers, more precisely those based on the earliest deadline first policy, are reported here (observations for static priority driven schedulings are very similar).

Aperiodic response time vs periodic load

In a first time, we shall analyze the mean response time of aperiodic requests as a function of the periodic load (more detailed analyses may be conducted, but we shall not elaborate on them here). For a given periodic load, 100 simulations are realized (to obtain a relatively stable mean response time). The periodic task set is composed of 5 to 10 tasks with periods ranging from 10 to 50 (bigger periods or systems lead to unacceptable computation times for the slack stealing algorithm) and late deadline \( (D_i = T_i) \). The \( C_i \)'s of periodic requests are chosen randomly and the task set is attributed to the corresponding periodic load.
class (with a granularity of 0.025). The server period is equal to 40 units of time and its capacity is equal to \((1 - U_p) \times 40\), such that the sum of periodic load and the server capacity is around 1. For aperiodic tasks, we generate randomly between 5 and 10 of them per 100 units of time of simulation, and the work generated by those requests per unit of time is approximately equal to 10%.

Figure 1 illustrates results obtained for the different servers; Figure 2 is a finest view without the background processing server and on a slightly shorter load range.

![Figure 1: Mean aperiodic response time vs periodic load.](image1)

![Figure 2: Mean aperiodic response time vs periodic load, a finest view.](image2)

Many observations can be made about this graphic. First, we can see that the performance of the background processing for small periodic load \((U_p < 0.7)\) is not so bad and comparable to other preserving bandwidth servers. But when the periodic load becomes more important, the performances of this scheduling technique deteriorates quickly.

We can also see that the polling server is quite bad (even worse than background processing for a small periodic load); but the extended polling server presents more interesting results than the basic definition of this server and its performances are comparable to other preserving bandwidth servers.

Another observation is that the preserving bandwidth servers present similar performances but may be ranked. After the polling server (with rather poor results) comes the extended polling server and the sporadic server. Next we have the priority exchange server and then the deferrable server, with very similar performances. Then comes the total bandwidth server, that presents results comparable to the slack stealing technique up to a periodic load of 85%. Finally comes the slack stealing which presents the best performances (in fact, it gives the optimal response time for aperiodic requests [7]), the improvement being especially sensible for high loads (above 85%);

this server requires an important amount of memory to work (and a lot of computations to be performed beforehand), with the consequence that a practical implementation is hard, even impossible for most systems.

In summary, we can categorize these techniques in three classes: the background processing that presents not so bad results for low periodic loads, the classical preserving bandwidth servers (the deferrable server, the priority exchange server and the sporadic server) which present good average performances and finally the total bandwidth server and the slack stealing that give very good results but the slack stealing, which is definitely better for high loads, requires an important (often unacceptable) amount of resources.

**Number of preemptions**

Let us now consider the number of preemptions for the various aperiodic servers: we assumed the switching times are negligible, but we know this may be wrong if there are too many preemptions. We shall use a simulation on 500 units of time. Periodic and aperiodic preemptions are undistinguishable in our study. Table 2 illustrates the results obtained on 100 simulations.

<table>
<thead>
<tr>
<th>Scheduling algorithms</th>
<th>average number of preemptions</th>
</tr>
</thead>
<tbody>
<tr>
<td>Background processing</td>
<td>61</td>
</tr>
<tr>
<td>Polling server</td>
<td>320</td>
</tr>
<tr>
<td>Deferrable server</td>
<td>349</td>
</tr>
<tr>
<td>Priority exchange server</td>
<td>316</td>
</tr>
<tr>
<td>Sporadic server</td>
<td>321</td>
</tr>
<tr>
<td>Total bandwidth</td>
<td>59</td>
</tr>
<tr>
<td>Slack stealing</td>
<td>57</td>
</tr>
</tbody>
</table>

Table 2: Number of preemptions on 500 units of time for the various scheduling algorithms.

Before we analyze the various results, we may men-
tardiness of aperiodic requests

First In, First Out (FIFO) is a simple and widely used scheduling policy. It prioritizes tasks based on their arrival time. However, FIFO can lead to the phenomenon of starvation, where some requests may wait indefinitely because they arrive after others with lower priority.

Aperiodic requests are those that do not have a fixed period of occurrence. For example, a user who submits a request and then waits for the result. These requests are typically handled by a first-in, first-out (FIFO) queue, where requests are processed in the order they arrive. However, FIFO can lead to significant delays for certain types of requests, especially those that are not time-critical.

To address this issue, various scheduling policies have been proposed. Some of these policies include:

1. Least Laxity First (LLF): This policy prioritizes requests based on their laxity, which is the difference between the deadline and the current time. Requests with the smallest laxity are processed first. This policy can be effective in reducing tardiness, especially in environments with a high number of aperiodic requests.

2. Earliest Deadline First (EDF): This policy prioritizes requests based on their deadlines. Requests with the earliest deadline are processed first. This policy can be effective in environments where deadlines are critical.

3. Round Robin (RR): This policy allocates a fixed time slice to each request in the queue. Once a request is granted the time slice, it is removed from the queue. RR can be effective in environments where fairness is important.

4. Priority Scheduling: This policy assigns a numerical priority to each request, and requests are processed in order of decreasing priority. This policy can be effective in environments where some requests are more important than others.

Policy to schedule aperiodic requests

The scheduling algorithms defined in section 3 and issued from the literature give the instants when the aperiodic requests receive the CPU but generally do not specify which aperiodic request must be served (first); generally the First In First Out rule is assumed (and was used to get the figures and the table of the previous sections) but we shall see that this choice is not necessarily the best if a soft deadline is specified for each aperiodic request. In fact, as we shall see, we can obtain with other policies a mean response time for aperiodic requests 50% less than with the First In First Out policy.

We shall here investigate the phenomenon for two other policies: Least Laxity First, which consists to favor the aperiodic requests who has the least laxity (i.e., the maximum time a task can be delayed on its activation to complete within its deadline) and Earliest Deadline First, which consists to favor the aperiodic request who has the nearest deadline. We shall compare these policies to the First In First Out rule. Other policies can be defined but are not considered in this paper.

Before we compare the different results obtained in the various simulations, it is interesting to analyze the complexity of the considered policies. The First In First Out strategy is very simple to implement and its time complexity is $O(1)$. The Earliest Deadline First and the Least Laxity First policies are more heavy to implement and require a time complexity in $O(m)$, if $m$ is the number of active aperiodic requests.

We note that these policies cannot be easily adapted to the total bandwidth server and the slack stealer.

Figure 3 presents the tardiness (i.e., the time a task stays active after its deadline) of aperiodic requests as a function of the aperiodic load (i.e., computation time demanded by aperiodic tasks in an interval, divided by the length of the interval) for the 3 studied policies, for the sporadic server (observations for other servers are similar). The periodic load and the server capacity are near to $\frac{1}{2}$, (the system is heavily loaded). For each aperiodic request, the deadline is chosen equal to the computation time ($p_i = d_i$).

We can see that differences between First In First Out and the other two policies are relatively important: when the aperiodic load increases, the tardiness of aperiodic requests is doubled with this first policy. Earliest Deadline First and Least Laxity First have similar performances in these conditions.

In Figure 4, the system is weakly loaded: the sum of the periodic (near to $\frac{1}{2}$) and the server capacity (also near to $\frac{1}{2}$) is around $\frac{3}{4}$. Results are now quite different: the tardiness of aperiodic requests is still doubled for high aperiodic loads with First In First Out than Earliest Deadline First, but now Least Laxity First is inferior to the Earliest Deadline First policy. In fact, the results of this policy take place between the others two.
6. Conclusions

In this paper we have described the most popular aperiodic servers issued from the literature. We have also presented a simulation tool used to compare the performances of these aperiodic servers. A comparison has been realized, with the use of metrics as varied as the response time, the number of preemptions and the tardiness. From the behaviour of the response time of aperiodic requests as a function of the periodic load, we have classified the performances of the various servers. But with the study of the number of preemptions, a more accurate classification is realized. We can synthesize the results of the comparison as follow. The background processing gives not so bad results when the periodic load is low ($U_P < 0.7$) but the performances of this server deteriorate very quickly when the periodic load increases. The preserving bandwidth servers give good average results but generate a great number of preemptions. The slack stealer gives, as expected the best results but cannot lead to practical implementations for large systems because this server has too important memory and time requirements. Finally, the best compromise seems to be given by the total bandwidth server, which has performances very close to the optimal, but for very high periodic loads, combined with a very easy, frugal and quick implementation.

We have seen that the various scheduling algorithms for mixed systems determine the instants where the aperiodic requests must be scheduled but generally do not specify which aperiodic request must be scheduled: the rule FIFO is assumed; we have shown the interest to consider the question, we have studied the performances of two other rules and have outlined the superiority of the EDF rule.

7. Acknowledgments

The authors wish to thank R. Devillers for his contributions.

8. References


