# HOLISTIC[1] SCHEDULABILITY ANALYSIS FOR DISTRIBUTED HARD REAL-TIME SYSTEMS

Ken Tindell[2]
Department of Computer Science
University of York, England

*ABSTRACT*

This report extends the current analysis associated with static priority pre-emptive based scheduling to address the wider problem of analysing schedulability of a distributed hard real-time system; in particular it derives analysis for a distributed system where tasks with arbitrary deadlines communicate by message passing and shared data areas. A simple TDMA protocol is assumed, and analysis developed to bound not only the communications delays, but also the delays and overheads incurred when messages are processed by the protocol stack at the destination processor. The report illustrates how a window-based analysis technique can be used to find the worst-case response times of a distributed task set. An extended example illustrating the application of the analysis is presented.

## 1. INTRODUCTION

A common way of constructing a hard real-time system is to compose the system from a number of hard real-time tasks dispatched according to static priorities. Analysis is done *a priori* to determine the worst-case response times of each of the tasks, and the system is only deployed if these response times meet the timing requirements of the system. Hitherto, this approach has not been applied well to distributed embedded hard real-time systems: the priority ceiling protocol, for example, has been successfully implemented only on a single-processor architecture (although some work on a less successful parallel implementation has been done). One of the barriers to applying fixed priority scheduling to distributed systems has been the lack of integrated schedulability analysis that can be applied *a priori* to bound the timing behaviour of a distributed system (including both processing delays and communications delays). This report is concerned with developing such analysis for a simple computational model (space considerations prohibit the addressing of a more complex, and powerful, computational model).

One of the most important problems with *a priori* analysis for distributed fixed priority systems has been the complications introduced by communications costs: the delays for messages being sent between processors must be accurately bounded, and the overheads due to communications must be strictly bounded. Other overheads also need to be addressed: for example, the overheads due to operating a so-called tick scheduler [18, 20] have been bounded accurately.

---

[1]Holism, *n*. the theory that the fundamental principle of the universe is the creation of wholes, *i.e.* complete and self-contained systems [From Chambers 20th Century Dictionary]

[2]The author can be contacted via e-mail as `ken@minster.york.ac.uk`

In this report we argue that a *holistic* approach to schedulability analysis for distributed systems must be taken; the schedulability analysis for single processor systems is integrated with timing analysis for hard real-time messages on a communications system to provide an overall piece of analysis. This analysis can be applied *a priori* to a particular configuration of a distributed hard real-time system to determine the timing bounds of the system as a whole (as will be discussed later, this analysis can be used as the basis for determining a suitable configuration).

Single processor schedulability analysis for fixed priority tasks has received considerable attention. In recent years the original fixed priority analysis [1] has been considerably extended, relaxing many of the assumptions of the original computational model. For example, Lehoczky *et al* [8] provided exact-case analysis to determine the worst-case timing behaviour of rate monotonic tasks. Leung and Whitehead [3] formulated an alternative priority assignment policy, where task deadlines can be less than the period of a task, and provided simple analysis to determine the schedulability of such tasks. Sha *et al* [9] discovered a concurrency control protocol to permit tasks to share critical sections of code. Audsley *et al* [11] permitted the addition of guaranteed sporadic tasks (where there is a minimum time between the re-arrivals of such tasks). Tindell *et al* [20] extended the approach further to characterise the re-arrival pattern, covering 'bursty' sporadic and periodic tasks, and introduced the concept of *release jitter* (where a task is not released into the system immediately upon arrival, but may suffer a bounded deferral time). Lehoczky [10] formulated the concept of a 'busy period', using this to provide qualitative analysis for tasks with arbitrary static deadlines. Lehoczky also indicated that neither the rate monotonic nor deadline monotonic priority assignment policies are optimal for such task sets. Tindell [18] provided quantitative analysis for arbitrary deadlined tasks with release jitter and bursty behaviour.

Schedulability analysis for communications is much less complete. Strosnider [6] applied the rate monotonic family of analysis to the 802.5 token-ring communications protocol to guarantee access times to the bus. Tindell *et al* [12] extended this work by showing how the general fixed priority analysis of Tindell *et al* [20] could be applied to the 802.5 protocol and to a generalised TDMA protocol. Crucially, the *end-to-end* deadlines of messages are taken into account: once a message arrives at the destination processor it must be processed and delivered to the destination application task.

In this report we will reproduce schedulability analysis for fixed-priority tasks with arbitrary deadlines, and then use this analysis to determine the worst-case response times of messages sent between processors. We will then extend the processor schedulability analysis to address the delivery costs of messages (both to bound the overheads on a destination processor, and to bound the delivery times of the messages, thus obtaining the end-to-end response times). The rest of this report is structured as follows: the next section introduces the basic single processor schedulability analysis used throughout this report, and gives the single-processor computational model. Section 3 derives the communications analysis for a simple TDMA protocol, and gives the communications model. Section 4 integrates the results of the processor and communications schedulability analysis to produce analysis for a distributed hard real-time system architecture. Section 5 gives an extended example, illustrating the application of the analysis using a simple tool. Section 6 summarises this report and offers conclusions. Section 7 contains a glossary of notation used throughout this report.

## 2. SINGLE PROCESSOR SCHEDULABILITY ANALYSIS

This section reproduces and decribes the schedulability analysis for arbitrary deadlined tasks. For simple periodic and sporadic fixed priority tasks [4, 20, 14] the following equation can be used to compute the worst-case response time of a given task $i$:

$$r_i = C_i + B_i + \sum_{\forall j \in hp(i)} \left\lceil \frac{r_i}{T_j} \right\rceil C_j \tag{1}$$

where $r_i$ is the worst-case response time of a given task $i$, $hp(i)$ is the set of all tasks of higher priority than task $i$ on the same processor, $C_i$ is the worst-case computation time of task $i$, and $T_j$ is the minimum time between successive arrivals of task $j$ (if $j$ is periodic then $T_j$ is equal to the period).

Priorities are assumed to be unique, and that tasks do not voluntarily suspend themselves (until they have completed their execution). The term $B_i$ is the longest time that task $i$ could be spent blocked by a lower priority task, and is computed according to the priority ceiling protocol analysis. For the purposes of this report we introduce the concept of a 'protected object', where a priority-ceiling semaphore is used to guard access to a Hoare monitor [2]. The concurrency control requirements of each task is then characterised by the objects and methods accessed by that task. The worst-case computation time of each method of each object is computed, which can then be used to both compute the blocking factor $B_i$ according to the priority ceiling protocol [9], and be used to compute the worst-case execution time of the caller tasks [15].

The worst-case response time of task $i$ is equal to the smallest value of $r_i$ that satisfies equation 1. This can be found by the following recurrence relation:

$$r_i^{n+1} = C_i + B_i + \sum_{\forall j \in hp(i)} \left\lceil \frac{r_i^n}{T_j} \right\rceil C_j \tag{2}$$

A suitable initial value for $r_i$ is 0.

Equation 2 is guaranteed to converge if the processor utilisation is ≤ 100% [4, 17]. The above analysis makes the assumption that the worst-case response time of a task $i$ must be less than $T_i$ (*i.e.* the deadline must be less than the period). The analysis also assumes that as soon as a task arrives it is released (*i.e.* immediately placed in a notional priority-ordered run-queue). If this is not the case (for example, if the task is logically able to run, but the system to which the task is allocated has not recognised this) then the analysis can be updated [20]:

$$r_i = J_i + w_i \tag{3}$$

where $w_i$ is given by:

$$w_i = C_i + B_i + \sum_{\forall j \in hp(i)} \left\lceil \frac{J_j + w_i}{T_j} \right\rceil C_j \tag{4}$$

The term $J_i$ is the worst-case delay between a task *arriving* (*i.e.* logically being able to run, yet not having been detected as runnable), and being *released* (*i.e.* placed in the run-queue), and is termed the *release jitter*. The effect of the deferral of the release of a task was first noted by Rajkumar with reference to the blocking of a task on an external event [7].

Release jitter is a problem because the worst-case time between successive *releases* of a task is shorter than the worst-case time between *arrivals* of a task. Consider the following scenario: a task *j*, of higher priority than task *i*, *arrives* at time 0. At a later time $J_j$ later task *j* is *released* (perhaps *j* is a sporadic which must be polled for by a tick scheduler). At the same time task *i* is released. Task *j* immediately pre-empts task *i*, as expected. At time $T_j$ task *j* re-arrives. This time *j* is immediately released (perhaps *j* arrived just before the tick scheduler polling period). From the view of task *i*, task *j* has arrived with time $T_j - J_j$ between arrivals (Figure 1).

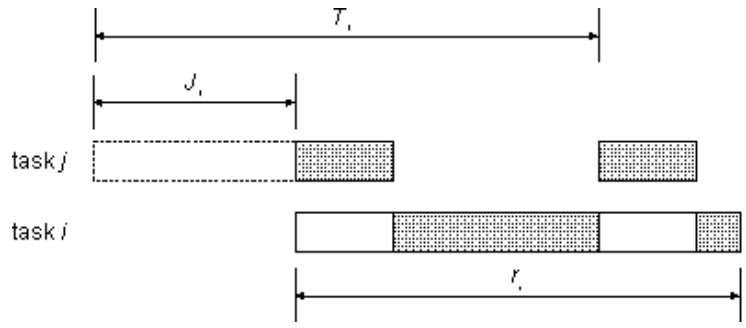


Figure 1: the problem of release jitter

Over a large number of periods task *j* will execute at the period $T_j$, but over a short period of time (between just two successive invocations of *j*) this rate is optimistic. The worst-case scheduling scenario for this short-term inter-arrival 'compression' is as described above: a task *j* is *released* at the same time as the level *i* busy period. The inclusion of the term $J_j$ in equation (4) accounts for this behaviour.

To avoid the problem that a later release of the same task could be delayed by the non-completion of the earlier release, we insist for the above equation that the time spent in the run-queue must be less than the task period (*i.e.* $w_i \le T_i$). To relax this $w_i \le T_i$ assumption we use the arbitrary deadline analysis developed by Tindell [18]:

$$r_i = \max_{q=0,1,2,\ldots} \left( J_i + w_i(q) - qT_i \right) \tag{5}$$

$$w_i(q) = (q+1)C_i + \sum_{\forall j \in hp(i)} \left\lceil \frac{J_j + w_i(q)}{T_j} \right\rceil C_j \tag{6}$$

The above equations are equivalent to equations (3) and (4) if $w_i \le T_i$. Again, the equations are guaranteed to produce a result if the processor utilisation is ≤ 100%. The sequence of values of *q* in equation (5) is finite since only values of *q* where $w_i(q) > (q+1)T_i$ need be considered.

A simple tool can be built which embodies the above equations and calculates *a priori* the worst-case response times of tasks on the processor. Such a tool is used to determine the worst-case timing characteristics of the example in section 5.

A common way of implementing a fixed priority scheduler is to use *tick based scheduling*: a timer interrupts the processor periodically and runs the scheduler. One of the activities of the scheduler is to make runnable (*i.e.* place in a notional priority-ordered run-queue) any periodic and sporadic tasks that have arrived since the last invocation of the scheduler (for example, the scheduler used by Locke *et al* [13]). A tick-scheduling approach is usually simple to implement, but can lead to pessimistic behaviour if poor analysis of the scheduler overheads is used. The overheads due to operating tick scheduling can be accurately determined by applying the same family of scheduling analysis as described in previous sections.

Before applying the analysis we will briefly describe some important characteristics of certain tick scheduling implementations. A common way of implementing tick scheduling is to use two queues: one queue holds a deadline ordered list of tasks which are awaiting their start conditions (such as a start time for periodics, or a start event — such as a value in an I/O register — for sporadics); we denote this queue the *pending queue*. The other queue is a priority-ordered list of runnable tasks, denoted the *run queue.* At each clock interrupt the scheduler scans the pending queue for tasks which are now runnable and transfers them to the run queue. Most implementations of such a queuing system have the following characteristic: the computation cost to take a task from the pending queue to the run queue is lower if more than one task is taken at the same time. For example, in one implementation of a run-time system at York [19] the worst-case cost to handle a timer interrupt is 66µs. The cost to take the first task from the pending queue is another 74µs (we denote this time $C_{QL}$). For each subsequent task removed (as part of the processing within the same interrupt) the cost is 40µs (we denote this time $C_{QS}$). These time savings are the result of one-off costs associated with setting up loops, *etc*. We now develop analysis to accurately account for these overheads.

The costs of the periodic timer interrupt can be modelled as a simple task, with worst-case computation time $C_{clk}$ and period $T_{clk}$; within a time window of width $w$ the worst-case number of timer interrupts is given by:

$$L = \left\lceil \frac{w}{T_{clk}} \right\rceil \tag{7}$$

Now, within the same window the worst-case number of times tasks move from the pending queue to the run queue is given from equation 6 by:

$$K = \sum_{\forall j \in hp(i)} \left\lceil \frac{J_j + w}{T_j} \right\rceil \tag{8}$$

If, over a window of width $w$, the total number of task queue moves, $K$, is less than the number of clock interrupts $L$, then in the worst-case all of the queue manipulations are full cost (*i.e.* each taking a worst-case computation time of $C_{QL}$), and the full cost of tick scheduling overheads is:

$LC_{clk} + KC_{QL}$

If, over $w$, $K$ is greater than $L$ then only the first $L$ task queue moves are at the full cost $C_{QL}$; the remaining $K - L$ require only $C_{QS}$ each. Hence the tick scheduling overheads for a task $i$ over a window of width $w$ are:

$$\tau_i(w) = LC_{clk} + \min(L, K)C_{QL} + \max(K - L, 0)C_{QS} \tag{9}$$

These overheads are additional computational interference, and thus equation (6) is updated to:

$$w_i(q) = (q+1)C_i + \sum_{\forall j \in hp(i)} \left\lceil \frac{J_j + w_i(q)}{T_j} \right\rceil C_j + \tau_i\left(w_i(q)\right) \tag{10}$$

## 3. COMMUNICATIONS SCHEDULABILITY ANALYSIS

This section applies the same family of analysis described in the previous section to the bounding of message delays across a communications system. We first describe the communications model assumed.

A number of processors are connected to a shared broadcast bus; for a processor to transmit on the bus it must have exclusive access to the bus. A number of protocols exist to arbitrate between processors when accessing a shared communications resource, but for the purposes of this report we assume that the TDMA protocol is used. The TDMA protocol dictates that each processor is able to transmit for a fixed time (idling the bus if no data is to be transmitted), termed a *slot*, before stopping and letting other processors transmit. The cycle of processors transmitting on the bus is fixed, and hence the cycle time of the bus, denoted $T_{TDMA}$ can be determined. In strict TDMA all processors have the same slot size. We relax this assumption, and permit each processor to have an independent slot size.

Messages are assumed to be broken up into packets by the sender task, with large messages requiring many packets. Each message is assigned a fixed priority, and all packets of the message are given this priority. Packets from the same message are queued in FIFO order, such that the last packet of a message to be transmitted corresponds to the last portion of the message. Packets are placed into a priority-ordered queue shared between the host processor and the network adapter responsible for physically transmitting the packets onto the network (the shared buffer could be implemented as dual-port memory). The packets are queued by the task sending the message, via a protected object guarding the queue; the overheads of packetising the message and queueing the packets are assumed to be included in the worst-case execution time of the sender task.

When a processor is allowed to transmit on the bus (indicated by a local clock synchronised to within $\Delta$ of a notional global time) the network adapter removes from the head of the packet queue all the packets that can be transmitted in a single slot, and then proceeds to transmit them, in priority order. In order to prevent clock drift leading to two processors transmitting at the same time the TDMA cycle time must be set up such that there is a $2\Delta$ time between the end of one processor transmitting and the start of the next processor transmitting ($2\Delta$ is the largest clock difference between any two local clocks).

A packet arriving at the destination processor either causes a packet interrupt to be raised, or an indication in a hardware status register to show the arrival. A packet delivery task is released (either directly by the packet interrupt, or indirectly by the tick-scheduler polling the hardware status register) to process the packet. The driver must remove the packet from the packet buffer shared between the network adapter and the host processor, and then place the packet in a temporary area where the message is re-assembled. If the arrived packet is the last packet of a message the destination task is then released to process the message.

We have the following restrictions on message passing: each sender task can send a fixed set of messages, each of bounded size, to fixed destination tasks. Each message $m$ in the set of messages a given task $i$ can send may be sent once every $n_m$ invocations of task $i$. Thus $m$ inherits a periodicity of $n_m T_i$ from the sender task $i$ (Figure 2).

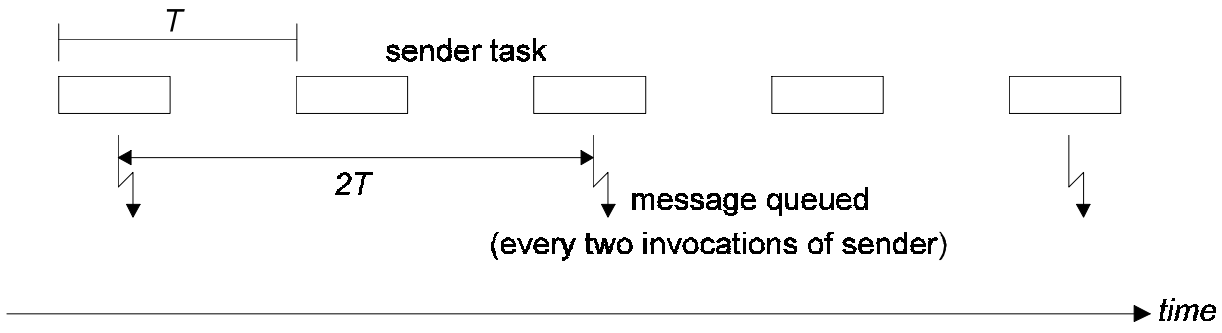


Figure 2: message queued every other invocation of sender and inherits a period of $2T$

In Figure 2, the message queued by the task has a value of $n$ of 2, and thus inherits a period of $2T$.

Each message must have a unique destination task, and no task can receive more than one message (if more than one message is destined for a certain task then extra 'server' tasks must be created which copy the message into a protected object which the destination task then reads). These restrictions are needed in order to bound the peak load on the communications bus, and (as will become clear later) enable the schedulability of the destination tasks to be determined.

We now derive the schedulability analysis to bound the time taken for a message $m$, sent from a processor $p$, to arrive at the destination processor. We introduce some notation: $P_m$ is the number of packets, of fixed size, that message $m$ is composed of. $T_m$ is the period of the message, inherited from the sender task $s(m)$. $S_p$ is the number of packets that processor $p$ is permitted to transmit in its TDMA slot.

Every $T_{TDMA}$, the communications adapter attached to the host processor takes $S_p$ packets from the head of the packet queue, and begins transmitting them (the $S_p$ packets can be considered to be instantaneously removed from the queue by the communications adapter). The number of TDMA cycles needed, in the worst-case, for all the packets of a message $m$ to be removed from the queue, is given by:

$$\left\lceil \frac{P_m + I_m}{S_p} \right\rceil \tag{11}$$

where $I_m(w)$ is the number of packets that can be queued ahead of $m$ (*i.e.* the number of higher priority packets) in the packet queue, in the worst-case. Therefore, the worst-case time that a message $m$ spends in the packet queue is given by:

$$\left\lceil \frac{P_m + I_m}{S_p} \right\rceil T_{TDMA} \tag{12}$$

The number of packets that can be queued ahead of message $m$ in a time $w$ can be found by realising that any higher priority message behaves in the worst-case as if it were queued periodically (with a period inherited from the sender task), but with an inherited release jitter. This release jitter is equal to the worst-case response time of the sender task: the task could queue a message at the last possible point during one invocation (*i.e.* just before finishing at the worst-case response time), and early during the subsequent invocation (*i.e.* just as it is released). In effect, the message can be considered to have 'arrived' as soon as the sender task arrives (since this is the earliest the message can be queued), but is deferred for any time up to the worst-case response time of the sender. This is illustrated in Figure 3.

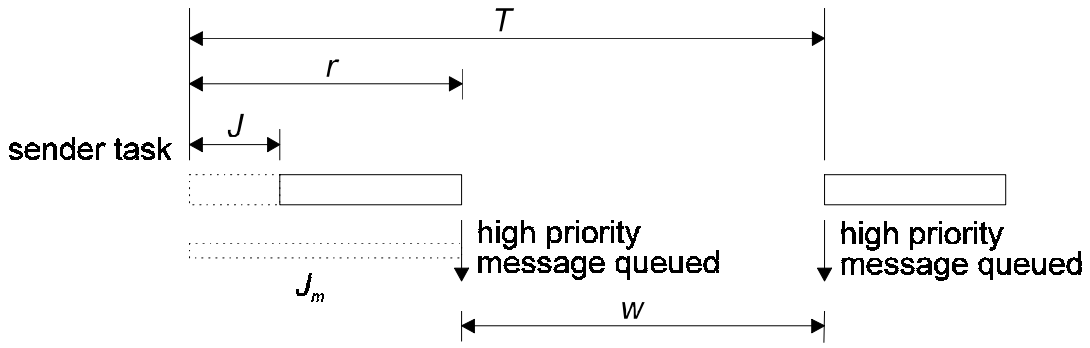


Figure 3: over the interval $w$ two high priority messages are queued

In Figure 3 the dotted boxes represent the worst-case release jitter of the sending task and the queued message; the high priority message has a release jitter of $J_m$ (equal to the worst-case response time of the sender task) and a period of $T$.

Therefore, the number of packets that can be queued ahead of message $m$ in a time $w$ is equal to:

$$I_m = \sum_{\forall j \in hp(m)} \left\lceil \frac{r_{s(j)} + w}{T_j} \right\rceil P_j \tag{13}$$

Where $T_j$ is given by:

$$T_j = n_{s(j)} T_{s(j)} \tag{14}$$

Equation 13 is almost identical to the interference part of the original scheduling equation (equation 4). Indeed, if $T_{TDMA}$ is set to 1 (representing the fact that the queue is serviced every time unit), $S_p$ is set to 1 (representing the fact that the server of the queue delivers a unit of service each iteration), and $P_m$ is set to $C_m$ (where $C_m$ represents the number of units of resource requested — time), then the original task scheduling equations can be established.

We can re-use the arbitrary deadline task schedulability analysis (equation 6) and apply it to message scheduling, giving the following message scheduling equations. The worst-case time message $m$ takes to arrive at the destination processor communications adapter is given (from equation 5) by:

$$a_m = \max_{q=0,1,2,3,\cdots} \left( w_m(q) + X_m(q) - qT_m \right) \tag{15}$$

Where the term $w_m(q) - qT_m$ gives the time the message $m$ spends in the packet queue, and $X_m(q)$ is the corresponding time taken to transmit the message from the communications adapter to the destination processor communications adapter, and $w_m(q)$ is given (from equations 6 and 12) by:

$$w_m(q) = \left\lceil \frac{(q+1)P_m + I_m\left(w_m(q)\right)}{S_p} \right\rceil T_{TDMA} \tag{16}$$

Where $I_m$, the number of packets ahead of message $m$, is given by:

$$I_m(w) = \sum_{\forall j \in hp(m)} \left\lceil \frac{w + r_{s(j)}}{T_j} \right\rceil P_j \tag{17}$$

To find $X_m(q)$ we need to know which of the $S_p$ packets transmitted in the last TDMA slot for message $m$ corresponds to the last packet of message $m$.

The maximum number of packets that need to be taken from the queue over the time $w_m(q)$ in order to guarantee the transmission of the last packet of $m$ is given from equations 16 and 17:

$$x = (q+1)P_m + I_m\left(w_m(q)\right) \tag{18}$$

The number of slots taken to transmit these packets, from equation 16 is:

$$s = \left\lceil \frac{x}{S_p} \right\rceil \tag{19}$$

The first $s - 1$ slot transmissions will not send the last packet of message $m$. In the final slot transmission, the last packet of message $m$ will be the $a$th packet of the final slot, where $a$ is given by:

$$a = x - (s-1)S_p \tag{20}$$

The time taken to transmit these packets is given by:

$$a\rho \tag{21}$$

where ρ is the time taken to transmit a packet on the communications bus. Allowing for a constant propagation delay we can therefore say that the last packet of message *m* arrives at the destination processor communications adapter at time:

$$X_m(q) = a\rho + \wp \tag{22}$$

after the packet is taken from the packet queue, where *s* and *x* are defined by equations 19 and 18 respectively.

Equation 15 gives the worst-case *arrival time* of a message *m*: the worst-case time between queueing the message *m* and it arriving in the packet buffer of the network adapter at the destination processor. To find the *end-to-end delay* — the worst-case time between queueing a message and it arriving at the destination task — we must determine the delivery delay.

Each packet is processed by a separate invocation of the packet handler (released either by the tick scheduler or by an interrupt). In the worst-case, the packet handler could be invoked once every ρ, since this is the fastest rate that packets can arrive from across the network to the processor. This is equivalent to assigning ρ to the period of the delivery task, $T_{deliver}$. A bound on the number of invocations of task *deliver* within a window of size *w* is therefore:

$$\left\lceil \frac{w + J_j}{\rho} \right\rceil \tag{23}$$

This bound can be pessimistic: in general it is possible that the worst-case computation time of the packet handler, $C_{deliver}$, could exceed the worst-case packet re-arrival time ρ, leading to processor with a utilisation > 100%. However, there is another bound on the number of packets arriving in a time window *w*: only a certain set of messages can be destined for a processor *p*. These messages are queued periodically with a period $T_m$ (inherited from the task sending *m*) and release jitter equal to $r_{s(m)} + a_m$. By using similar analysis to that of equation 17 we can say that the number invocations of the packet handler is bounded by:

$$l_p(w) = \sum_{\forall k \in incoming(p)} \left\lceil \frac{w + r_{s(k)} + a_k + J_{deliver}}{T_k} \right\rceil P_k \tag{24}$$

Where *incoming*(*p*) is the set of messages in the system destined for processor *p*, $r_{s(k)}$ is the worst-case response time of the task sending message *k* (equation 5), $a_k$ is the worst-case delay between queueing message *k* and message *k* arriving at processor *p* (equation 15), and $T_k$ is the inherited period of message *k* (equation 14).

The lower bound of the two bounds (equations 23 and 24) on the number of invocations can be used, and hence the number of packets arriving at processor *p* over a time window *w* is bounded by:

$$v_{deliver}(w) = \min\left( l_p, \left\lceil \frac{w + J_j}{\rho} \right\rceil \right) \tag{25}$$

In equation 6 the term $(q + 1)C_i$ represents the worst-case computation due to task $i$. When calculating the worst-case response time of task *deliver* the term $(q + 1)C_i$ can be replaced with:

$$u_{deliver,q}(w) = \min\left(l_p(w), q+1\right)C_{deliver} \tag{26}$$

where $l_p(w)$ is the bound on the number of packets coming into processor $p$ in a time window of size $w$ (equation 24)

Thus the equation for $w_{deliver}(q)$ can be re-written (from equations 26 and 10):

$$w_i(q) = u_{i,q}\left(w_i(q)\right) + \sum_{\forall j \in hp(i)} \left\lceil \frac{J_j + w_i(q)}{T_j} \right\rceil C_j + \tau_i\left(w_i(q)\right) \tag{27}$$

where $i = deliver$

Since packets are serviced in FIFO order the delivery of the last packet of message $m$ entails the delivery of message $m$. Therefore, the delivery time of message $m$ is simply the worst-case time to deliver a packet, *i.e.* $r_{deliver}$. Hence the end-to-end response time of a message $m$ can be bounded by summing the worst-case arrival time of a message $m$ and the worst-case delivery time (equations 15 and 27):

$$r_m = a_m + r_{deliver} \tag{28}$$

where *deliver* is the delivery task allocated to the destination processor of message $m$.

## 4. HOLISTIC SCHEDULING THEORY

The previous two sections have derived *a priori* analysis that gives the worst-case timing characteristics of a given task on a given processor, and a given message sent to a given processor. In this section we show how the analysis can be integrated to provide a powerful model of distributed hard real-time computation.

The most important aspect of integrating the processor and communications schedulability analysis is to bound the overheads due to packet handling on given processor. Equation 25 gives the worst-case number of packets arriving at a given processor in a given time window, and this is used to bound the worst-case response time of the delivery task (equations 5 and 27). This approach can be extended to bound the interference from the packet handler on lower priority tasks (from equation 6):

$$w_i(q) = (q+1)C_i + \tau_i\left(w_i(q)\right) + \sum_{\forall j \in hp(i)} \begin{cases} v_j\left(w_i(q)\right) & \text{if } j = deliver \\ \left\lceil \dfrac{J_j + w_i(q)}{T_j} \right\rceil C_j & \text{otherwise} \end{cases} \quad (29)$$

Note that if there are no packets to send in a slot then it is not permissable to send soft-real time packets instead of idling the bus *unless* these soft real-time packets are added to the set *incoming*(*p*): in most communications hardware architectures the communications adapter is unable to distinguish between hard and soft real-time packets, and will generate an interrupt regardless of the class. Hence if more packets arrive at a processor than were allowed for, too many interrupts would be generated, potentially causing missed deadlines.

When a message fully arrives at a destination processor, the destination task is released, and can then read the message. The destination task inherits a release jitter in the same way that a message inherits release jitter from the sender task. Thus, for a task $i$ that receives a message $m$, the inherited release jitter is equal to:

$$r_{s(m)} + a_m + r_{deliver} \quad (30)$$

where $s(m)$ is the task that sends the message $m$, and *deliver* is the packet handler on the destination processor of message $m$

This release jitter is in addition to any other jitter that the task may already have (for example, if the arrival of the task is polled for by the scheduler then it will have a release jitter of $T_{tick}$). So, for a task $d(m)$ (the destination of message $m$), with an existing jitter of $T_{tick}$ due to the operation of a tick scheduler, we have:

$$J_{d(m)} = r_{s(m)} + a_m + r_{deliver} + T_{tick} \quad (31)$$

We now indicate an interesting property of the scheduling equations. The scheduling equations are mutually dependent: the release jitter of a receiver task (equation 31) depends on the arrival time of the message $a_m$ (equation 15), which in turn depends on the interference from higher priority messages (equation 16), which in turn depends on the release jitter of sender tasks (equation 13). Hence the holistic scheduling equations cannot be trivially solved.

A solution to this problem can be found by realising that all of the scheduling equations are monotonic in window size $w$, response time, and release jitter (*i.e.* increasing any of these variables cannot lead to a decrease in any of the other variables). Therefore, it is possible to for a recurrence relation and iterate to a solution in the same way as it is possible to iterate to solve the scheduling equations (for example, equation 2): in the first iteration of the scheduling equations we set the inherited release jitter for all tasks to zero. On the $n$th iteration the inherited release jitter values can be set according to the results of solving the scheduling equations in the $(n-1)$th iteration.

One of the restrictions of the computational model is that task access to a protected object must always be local: it is not permitted to lock a semaphore on another processor. One way around this potential problem is to use an approach akin to RPC (Figure 4).

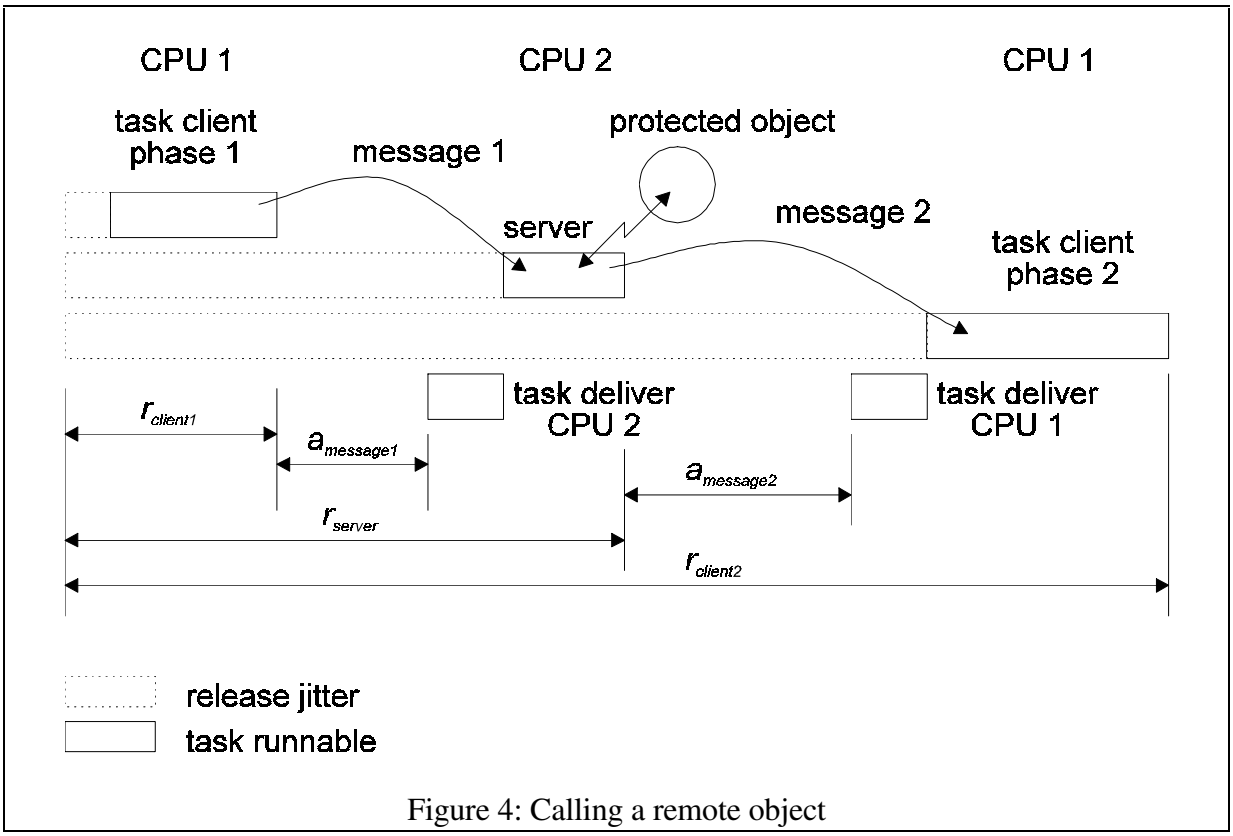


Figure 4: Calling a remote object

The sender task is split into two separate tasks: the first task sends a message to the processor where the object is stored, to a server task. The server task is activated once the message arrives, and calls the appropriate protected object method. The task then sends a return message to the second task, containing the relevant data. This second task then processes the message data as normal. The release jitter for the second task can become quite large, but there is a technique for keeping the release jitter small; this will be discussed in section 5. Note that this approach to accessing remote objects is very similar to that taken by Rajkumar *et al* [7], except that we assume the client task is split into two separate tasks; Rajkumar *et al* permit a single client task which suspends itself awaiting the reply from the remote server.

Another restriction of the model is that no task can receive more than one message. This is because of the difficulty in assessing which message should trigger the task. However, multiple messages sent to the same task can be permitted using a different mechanism: a server task is created for each message destined for task *i*, which then simply writes the message to a shared protected object. Task *i* can poll the protected object and read any messages. This has the advantage that task *i* and the tasks sending the message may share an *offset* [17] relationship (*i.e.* task *i* always arrives a fixed time after the arrival of a given sender task, using a global time mechanism to ensure all processor clocks are kept synchronised [5]) such that all messages for task *i* from the sender tasks are always guaranteed to have been delivered by the time task *i* arrives (Figure 5).

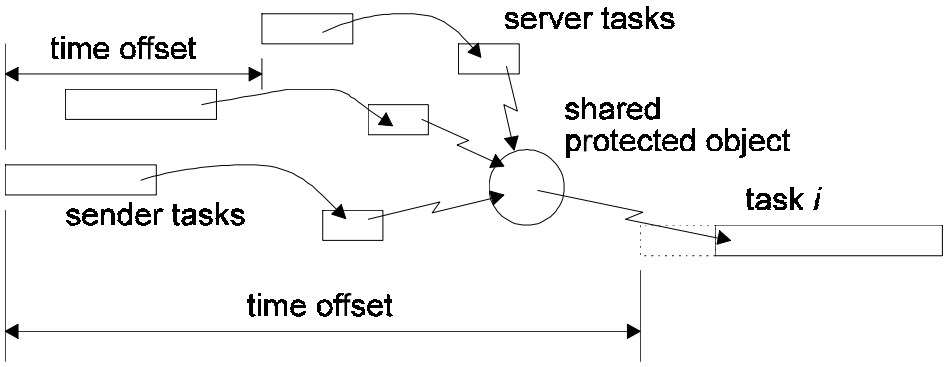


Figure 5: Sending multiple messages to a single task

To illustrate some of these principles we now give an extended example, using the analysis developed.

## 5. EXTENDED EXAMPLE

The example system consists of three processors, with 32 tasks and 7 objects. A total of 14 different messages are sent by tasks in the system, of which 13 require transmission across a shared communications bus. The packet size is set to be 1024 bytes, with a packet transmission time ρ of 800μs. The slot sizes of processors 1, 2, and 3 are 1, 1, and 3 packets respectively. The processor clocks are synchronised to global time to within 40μs, and hence the TDMA cycle time, $T_{TDMA}$, is 4240μs. The constant propagation delay $\wp$ is assumed to be 1μs. The clock tick period $T_{clk}$ is 1000μs, $C_{clk}$ is 66μs, $C_{QL}$ is 74μs, and $C_{QS}$ is 40μs.

The tasks represent the computation required for a hypothetical aircraft control system. Processor 3 represents a sensor processor, and has only 3 tasks allocated to it. These tasks monitor sensors and transmit the results to processors 1 and 2. Computation on processors 1 and 2 result in commands being sent to actuators, which are connected to processor 2.

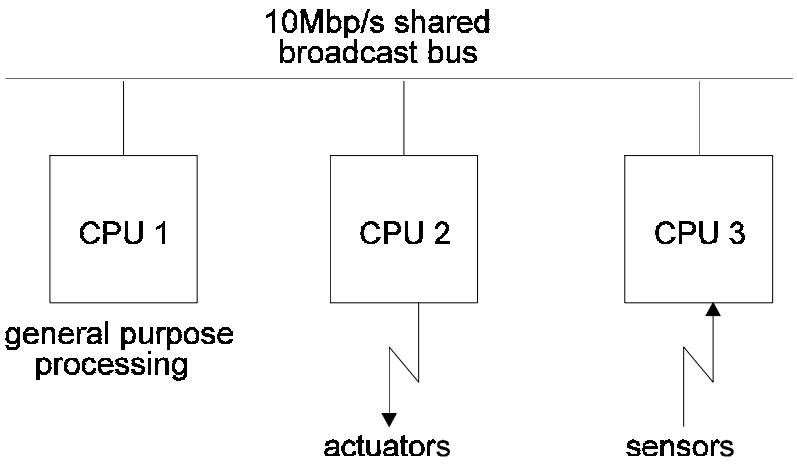


Figure 6: Architecture of the Example System

The following tables detail the timing requirements placed on the tasks in the system, and show the worst-case response times calculated according to the equations given in this report (a small tool was written to perform the calculations and print the results).

The timing characterstics of tasks allocated to processor 1 are given below:

| *Problem variables* | | | | *Computed values* | | |
|---|---|---|---|---|---|---|
| *task name* | *T* | *C* | *D* | *B* | *J* | *r* |
| deliver_cpu1 | 800 | 150 | X | 0 | 0 | 970 |
| task1 | 200000 | 2277 | 5000 | 0 | 0 | 4557 |
| deliver_air_fuse_data | 40000 | 420 | 15000 | 321 | 8890 | 14478 |
| deliver_air_data_update | 20000 | 552 | 20000 | 321 | 10685 | 17305 |
| deliver_air_data | 20000 | 496 | 20000 | 321 | 9885 | 17291 |
| task3 | 25000 | 1423 | 12000 | 354 | 0 | 11150 |
| task5 | 50000 | 3096 | 50000 | 354 | 0 | 15786 |
| task7 | 59000 | 7880 | 59000 | 354 | 0 | 27469 |
| task9 | 50000 | 1996 | 100000 | 354 | 15786 | 46679 |
| deliver_radar | 100000 | 3220 | 100000 | 354 | 34358 | 72237 |
| deliver_radar_update | 100000 | 3220 | 100000 | 343 | 55558 | 97046 |
| client1 | 200000 | 520 | 100000 | 343 | 0 | 42108 |
| client2 | 200000 | 1120 | 200000 | 343 | 107210 | 150538 |
| task11 | 1000000 | 954 | 200000 | 343 | 141521 | 185903 |
| task13 | 200000 | 1124 | 200000 | 343 | 0 | 49266 |
| task15 | 200000 | 3345 | 200000 | 343 | 0 | 74284 |
| task17 | 1000000 | 1990 | 1000000 | 0 | 0 | 77626 |

For processor 2:

| *Problem variables* | | | | *Computed values* | | |
|---|---|---|---|---|---|---|
| *task name* | *T* | *C* | *D* | *B* | *J* | *r* |
| deliver_cpu2 | 800 | 150 | X | 0 | 0 | 770 |
| task4 | 40000 | 996 | 14000 | 343 | 0 | 2879 |
| deliver_health | 100000 | 550 | 20000 | 343 | 16379 | 19998 |
| task2 | 25000 | 689 | 5000 | 343 | 0 | 4598 |
| task6 | 50000 | 4967 | 50000 | 410 | 0 | 10992 |
| server | 200000 | 2342 | X | 756 | 74359 | 88479 |
| task8 | 80000 | 9125 | 80000 | 756 | 0 | 24177 |
| task10 | 100000 | 5120 | 115000 | 756 | 0 | 30456 |
| deliver_actr | 200000 | 654 | 200000 | 756 | 59317 | 90493 |
| task12 | 1000000 | 3145 | 200000 | 350 | 96157 | 130270 |
| task14 | 25000 | 2325 | 200000 | 350 | 38161 | 74731 |
| task16 | 1000000 | 1455 | 1000000 | 0 | 83437 | 129891 |

For processor 3:

| *Problem variables* | | | | *Computed values* | | |
|---|---|---|---|---|---|---|
| *task name* | *T* | *C* | *D* | *B* | *J* | *r* |
| send_air | 20000 | 2245 | 20000 | 0 | 0 | 2665 |
| send_health | 100000 | 2322 | 100000 | 0 | 0 | 5185 |
| send_radar | 100000 | 12224 | 100000 | 0 | 0 | 18267 |

The symbol X under the deadline column in the above tables denotes “don't care” — that there is no intrinsic deadline for the task (although the response time of the task adds to the response

time of some other task with a deadline). Tasks *task11* and *server* are sporadic tasks with their release polled for by the tick scheduler, and hence have a base release jitter of $T_{tick}$. All other tasks have a base release jitter of zero. The calculated release jitter values in the above tables include the base release jitter values.

The messages sent by each task are described in the table below. The messages are listed in priority order. Note that message priorities between messages sent from different processors are meaningless, since the TDMA protocol only uses priority to arbitrate between messages sent from the same processor.

| *Problem variables* | | | | | | | *Computed values* | |
|---|---|---|---|---|---|---|---|---|
| *message name* | *size* | *source task* | *source CPU* | *destination task* | *destination CPU* | *n* | *packets* | *r* |
| air_data | 220 | send_air | cpu3 | deliver_air_data | cpu1 | 1 | 1 | 6811 |
| air_data_update | 550 | send_air | cpu3 | deliver_air_data_update | cpu1 | 8 | 1 | 7611 |
| health_data | 2230 | send_health | cpu3 | deliver_health | cpu2 | 1 | 3 | 10851 |
| radar_data | 2048 | send_radar | cpu3 | deliver_radar | cpu1 | 1 | 2 | 16091 |
| radar_data_update | 16384 | send_radar | cpu3 | deliver_radar_update | cpu1 | 8 | 16 | 37291 |
| message1 | 500 | task17 | cpu1 | task16 | cpu2 | 1 | 1 | 5811 |
| message2 | 800 | task13 | cpu1 | deliver_actr | cpu2 | 1 | 1 | 10051 |
| message3 | 779 | task4 | cpu2 | deliver_air_fuse_data | cpu1 | 1 | 1 | 6011 |
| message4 | 225 | task5 | cpu1 | task9 | cpu1 | 1 | 1 | 0 |
| message5 | 1255 | task17 | cpu1 | task12 | cpu2 | 2 | 2 | 18531 |
| message6 | 1255 | task3 | cpu1 | task14 | cpu2 | 7 | 2 | 27011 |
| message7 | 800 | task12 | cpu2 | task11 | cpu1 | 1 | 1 | 10251 |
| toserver | 500 | client1 | cpu1 | server | cpu2 | 1 | 1 | 31251 |
| fromserver | 1500 | server | cpu2 | client2 | cpu1 | 1 | 2 | 18731 |

As can be seen from the table, the radar update message is a large message, sent infrequently, at the lowest priority (of messages sent by processor 3). The worst-case response time of the message is 37291 µs. Message 4 in the table is a message sent between two tasks on the same processor, and thus the worst-case response time of the message is deemed to be zero. This results in an inherited release jitter for task 9, equal to the worst-case response time of task 5.

Details of the objects in the system are shown in the table below:

| *object name* | *ceiling task* | *host* | *object type* |
|---|---|---|---|
| messages_cpu1 | task3 | cpu1 | message_mgmt_object |
| messages_cpu2 | task4 | cpu2 | message_mgmt_object |
| messages_cpu3 | send_air | cpu3 | message_mgmt_object |
| air_data | deliver_air_fuse_data | cpu1 | air_data_object |
| gyro_data | task9 | cpu1 | gyro_data_object |
| actuator_ctrl | task6 | cpu2 | actuator_ctrl_object |
| radar_data | task3 | cpu1 | radar_data_object |
| health_data | server | cpu2 | health_data_object |
| buffer_mgmt_cpu1 | task13 | cpu1 | buffer_mgmt_object |
| buffer_mgmt_cpu2 | task12 | cpu2 | buffer_mgmt_object |

Details of the objects called are shown in the table below:

| *task name* | *object called* | *method called* |
|---|---|---|
| server | health_data | read_data |
| deliver_radar | radar_data | write_data |
| deliver_radar_update | radar_data | write_data |
| deliver_air_data | air_data | write_data |
| | | fuse_data |
| deliver_air_data_update | air_data | write_data |
| task12 | buffer_mgmt_cpu2 | enter |
| task13 | buffer_mgmt_cpu1 | enter |
| task9 | gyro_data | update |
| | radar_data | write_data |
| | air_data | write_data |
| | | read_data |
| | | fuse_data |
| deliver_air_fuse_data | air_data | fuse_data |
| task15 | gyro_data | calibrate |
| task10 | actuator_ctrl | set_ctrl |
| task6 | actuator_ctrl | set_ctrl |
| task3 | radar_data | read_data |
| task12 | health_data | update_health |
| task16 | health_data | read_health |
| deliver_actr | actuator_ctrl | set_ctrl |
| deliver_air_data | air_data | write_data |

Additionally, all tasks that send messages call method ‘queue_packet’ of the object of type ‘message_mgmt_object’ allocated to the processor.

Details of the worst-case execution times of each of the methods of each object are given in the table below:

| *object type* | *method name* | *WCET* |
|---|---|---|
| air_data_object | write_data | 256 |
| | read_data | 209 |
| | fuse_data | 321 |
| gyro_data_object | update | 221 |
| | calibrate | 252 |
| actuator_ctrl_object | set_ctrl | 410 |
| radar_data_object | read_data | 108 |
| | write_data | 354 |
| health_data_object | update_health | 756 |
| | read_health | 350 |
| buffer_mgmt_object | enter | 477 |
| | remove | 632 |
| message_mgmt_object | queue_packet | 343 |

The tasks *client1*, *server*, and *client2* illustate access to a remote object: *client1* sends a message to task *server*, which then accesses the object *health_data* (via the method *read_data*). The results are sent back to *client2* via a message (denoted *fromserver*, taking 18731 μs to arrive at the *client2* task. As can be seen, *client2* inherits a large release jitter (107210 μs) equal to the worst-case response time of *server* plus the time for message

*fromserver* to be sent to *client2*. The worst-case response time of task *server* itself is formed partly from release jitter inherited from *client1* and message *toserver*.

A large part of the release jitter for *client2* and *server* could be removed by using time offsets [17] between the three tasks. Time offsets ensure that the early arrival of a message does not entail an early release of the task waiting for the message. The reduction (or removal) of release jitter from a higher priority task using offsets does not improve the worst-case response time of that task, but will lead to a reduction in the worst-case response times of lower priority tasks.

## 6. SUMMARY AND CONCLUSIONS

This report has shown how to analyse distributed hard real-time systems conforming to a particular architecture — simple fixed priority scheduling of processors, with a simple TDMA protocol arbitrating access to a shared broadcast bus. The software architecture is a simple one, with periodic and sporadic tasks communicating via messages and shared data areas. The analysis has been applied to an extended example to illustate some of the ways a system can be composed, and some of the timing characteristics that result. The real benefit of the analytical approach taken is this report is not just to obtain *a priori* schedulability guarantees across a distributed system, but to aid the configuration of such a system. Tindell *et al* show how rudimentary analysis is used as part of a combinatorial optimisation algorithm to allocate tasks across a distributed system [16]; research in progress is applying the analysis of this report to the wider configuration problem.

## 7. GLOSSARY OF NOTATION

The following table summarises the notation used throughout this report.

| | |
|---|---|
| $T_i$ | Minimum time between arrivals of task $i$ |
| $C_i$ | Worst-case computation time of task $i$ |
| $B_i$ | Worst-case blocking time of task $i$ |
| $incoming(p)$ | Set of messages destined for processor $p$ |
| $hp(i)$ | Set of tasks of higher priority than task $i$ and located on the same processor |
| $J_i$ | Worst-case time between arrival and release for an invocation of task $i$; termed the *release jitter* |
| $r_i$ | Worst-case response time of task $i$, measured from the arrival of the task to the completion of the task |
| $n_m$ | Message $m$ will be sent once every $n_m$ invocations of the sender of the message |
| $s(m)$ | The sender task of message $m$ |
| $d(m)$ | The destination task of message $m$ |
| $P_m$ | The number of packets of which message $m$ consists |
| $\wp$ | The propagation delay across the broadcast bus |
| $\rho$ | The time taken to transmit a packet |
| $S_p$ | The number of packets a processor $p$ may transmit in its TDMA slot |
| $T_{TDMA}$ | The TDMA cycle time |
| $\Delta$ | The worst-case difference in time between any local clock and global time |

| | |
|---|---|
| $a_m$ | The worst-case arrival time of a message *m*, measured relative to the time the message is queued |
| $X_m$ | The worst-case time taken to transmit the last packet of message *m* once the TDMA slot for the processor has arrived |
| $r_{deliver}$ | The worst-case delivery time of a packet once arrived at the destination processor |
| $w_i$ | The time the processor spends executing tasks of priority equal or higher than that of task *i* |
| $C_{OL}$ | The largest computation time cost of taking a task from the pending queue to the delay queue |
| $C_{OS}$ | The per-task cost of taking more than one task from the pending queue to the delay queue |
| $C_{clk}$ | The cost of handling the timer interrupt driving the scheduler |
| $T_{tick}$ | The period of the timer interrupt that drives the scheduler |
| $I_m$ | The interference from higher priority messages on message *m* |
| $L$ | Defined by equation 7 |
| $K$ | Defined by equation 8 |
| $s$ | Defined by equation 19 |
| $l_p$ | Defined by equation 24 |
| $v_{deliver}$ | Defined by equation 25 |
| $u_{deliver}$ | Defined by equation 26 |

## 8. REFERENCES


[1] "Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment", C. L. Liu, J. W. Layland, *Journal of the ACM*, **20**(1), 1973, pp. 46-61

[2] "Monitors — An Operating System Structuring Concept", C.A.R. Hoare, *CACM,* 17(10), pp. 549-557, October 1974

[3] "On The Complexity of Fixed-Priority Scheduling of Periodic Real-Time Tasks", J. Y. T. Leung, J. Whitehead, *Performance Evaluation*, **2**(4), Dec 1982, pp. 237-250

[4] "Finding Response Times in a Real-Time System", M. Joseph, P. Pandya, *BCS Computer Journal*, **29**(5), Oct 86, pp. 390-395

[5] "Clock Synchronisation in Distributed Real-Time Systems", H. Kopetz, W. Ochsenreiter, *IEEE Transactions on Computers*, **C-36**(8), August 1987

[6] "Advanced Real-Time Scheduling Using the IEEE 802.5 Token Ring", J. K. Strosnider, T. Marchok, J. Lehoczky, *Proceedings of the IEEE Real-Time Systems Symposium*, Huntsville, Alabama, USA, 1988, pp. 42-52

[7] "Real-Time Synchronization Protocols for Multiprocessors", R. Rajkumar, L. Sha, J. P. Lehoczky, *Proceedings of the IEEE Real-Time Systems Symposium*, Huntsville, Alabama, 1988, pp. 259-269

[8] "The Rate Monotonic Scheduling Algorithm: Exact Characterisation and Average Case Behaviour", J. Lehoczky, L. Sha, Y. Ding, *Proceedings of the Real-Time Systems Symposium*, 1989

[9] "Priority Inheritance Protocols: An Approach to Real-Time Synchronisation", L. Sha, R. Rajkumar, J. P. Lehoczky, *IEEE Transactions on Computers*, **39**(9), September 1990, pp. 1175-1185

[10] "Fixed Priority Scheduling of Periodic Task Sets With Arbitrary Deadlines", J. P. Lehoczky, *Proceedings 11th IEEE Real-Time Systems Symposium*, Lake Buena Vista, Florida, USA, 5-7 Decmeber 1990, pp. 201-209

[11] "Hard Real-Time Scheduling: The Deadline Monotonic Approach", N. C. Audsley, A. Burns, M. F. Richardson, A. J. Wellings, *Proceedings 8th IEEE Workshop on Real-Time Operating Systems and Software*, 15-17 May 1991, Atlanta, GA, USA

[12] "Guaranteeing Hard Real Time End-to-End Communications Deadlines", K. W. Tindell, A. Burns, A. J. Wellings, Department of Computer Science, University of York, Report number RTRG/91/107, Dec 1991

[13] "Building a Predictable Avionics Platform in Ada: A Case Study", C. D. Locke, D. R. Vogel, T. J. Mesler, *Proceedings of the 12th Real Time Systems Symposium,* pp. 181-, December 1991

[14] "Applying New Scheduling Theory to Static Priority Pre-emptive Scheduling", N. Audsley, A. Burns, M. Richardson, K. Tindell, A. Wellings, Report RTRG/92/120 Department of Computer Science Report, University of York, February 1992

[15] "Implementation of the Worst-Case Execution Time Analyser", C. H. Forsyth, Task 8 Volume E, Deliverable on ESTEC Contract 9198/90/NL/SF, York Software Engineering Limited, University of York, June 1992

[16] "Allocating Real-Time Tasks (An NP-Hard Problem made Easy)", K. Tindell, A. Burns, A.J. Wellings, *Real-Time Systems* **4**(2), June 1992, 145-165

[17] "Using Offset Information to Analyse Static Priority Pre-Emptively Scheduled Task Sets", K. Tindell, Department of Computer Science Report, University of York, YCS 182, August 1992

[18] "An Extendible Approach for Analysing Fixed Priority Hard Real-Time Tasks", K. Tindell, Department of Computer Science Report, University of York, Report YCS189, December 1992

[19] "The Olympus Attitude and Orbital Control System: A Case Study in Hard Real-time System Design and Implementation", A. Burns, A.J. Wellings, C.M. Bailey, E. Fyfe, YCS 190, Department of Computer Science, University of York, 1993

[20] "Applying New Scheduling Theory To Static Priority Pre-emptive Scheduling", N. Audsley, A. Burns, K. Tindell, M. Richardson, A. Wellings, *Software Engineering Journal* (to appear)