Non-Preemptive Scheduling of Messages on Controller Area Network for Real-Time Control Applications*

Khawar M. Zuberi and Kang G. Shin

Real-Time Computing Laboratory
Department of Electrical Engineering and Computer Science
The University of Michigan
Ann Arbor, MI 48109-2122
{*zuberi,kgshin*}*@eecs.umich.edu*

Abstract

Scheduling messages on the Controller Area Network (CAN) corresponds to assigning identifiers (IDs) to messages according to their priorities. If fixed-priority scheduling such as deadline monotonic (DM) is used to calculate these priorities, then in general, it will result in low schedulability. Dynamic scheduling schemes such as earliest-deadline (ED) can give greater schedulability, but they are not practical for CAN because if the ID is to reflect message deadlines then a long ID must be used. This increases the length of each message to the point that ED is no better than DM. Our solution to this problem is the mixed traffic scheduler (MTS), which is a cross between ED and DM, and provides high schedulability without needing long IDs. Through simulations, we compare the performance of MTS with that of DM and ED (an imaginary scheduler which works like ED, except it needs only short IDs). We use a realistic workload in our simulations based on messages typically found in computer-integrated manufacturing. Our simulations show that MTS performs much better than DM and at the same level as ED*, except under high loads and tight deadlines, when ED* is superior.*

1 Introduction

Real-time controllers are today being used in diverse applications such as manufacturing automation, robotic manipulators, and antilock braking systems. Because of the nature of the controlled process and/or for fault-tolerance reasons, a real-time controller is built with a network of processors which must coordinate their efforts to achieve a common goal. For cost-effectiveness reasons, usually a bus topology is used to connect the processors.

In general, a real-time controller must detect events and then respond to them by taking appropriate actions. Real-world events are aperiodic in nature, but if an event occurs every 10–15ms on average and some action must be taken within 10ms of that event, then the corresponding sensor may be periodically polled to detect that event (with a period of 10ms, if processing time is neglected). But if the same event is known to occur *sporadically* and has a large minimum interarrival time (say 10s), then sampling the sensor every 10ms makes no sense — it is a waste of communication and computation bandwidth. A better solution will be to distribute intelligence by using *smart sensors* (sensors with limited processing abilities) [1]. In other words, it is far better to let the sensor detect the event and notify the processor. This way, instead of one message every 10ms, there will be just one message whenever the event occurs, which is at least 10s apart.

However, there is a disadvantage in using the above paradigm: sporadic messages are non-deterministic, making message scheduling much harder. But the expected advantages in terms of fewer messages motivate us to find a communication protocol which can efficiently handle sporadic as well as periodic messages.

Consider the kind of communication protocol which must be used to allow the smart sensor task to do its job. One can immediately think of at least two requirements:

R1. The protocol must give fast bus access to high-priority sporadic messages.

R2. Since all real-time systems also have periodic messages, the protocol should be able to support such messages efficiently as well.

When trying to find such a protocol, we can immediately disregard CSMA-CD protocols like Ethernet [2] because of their unbounded network access latency. We can consider synchronous protocols like the *token bus* [2] and the *time-triggered protocol* (TTP) [3]. They satisfy the second requirement but not the first: it is well-known that these protocols give a *large* worst-case bus access latency making them unsuitable for systems with tight-deadline sporadic messages [4].

*The work reported in this paper was supported in part by the NSF under Grants MIP-9203895 and DDM-9313222, and by the ONR under Grant N00014-94-1-0229. Any opinions, findings, and conclusions or recommendations are those of the authors and do not necessarily reflect the views of the funding agencies.

SOF	Identifier	Control	Data	CRC	Ack	EOF

SOF: Start of Frame
CRC: Cyclic Redundancy Code
EOF: End of Frame

Figure 1: Various fields in the CAN data frame.

The *Controller Area Network* (CAN) [5, 6] is a contention-based multi-master network which uses a wired-OR (or wired-AND) bus and has the potential to efficiently handle both periodic as well as sporadic messages, thus satisfying both **R1** and **R2**.

In this paper we explore the capabilities of CAN in carrying *mixed* periodic, sporadic, and non-real-time traffic. We first develop a scheduling algorithm called the *mixed traffic scheduler* (MTS) which handles messages of different priorities, and then evaluate its performance through simulations.

The next section describes the CAN protocol in detail. Section 3 describes the various types of messages in our target application workload. Section 4 gives the MTS algorithm and its schedulability conditions. Section 5 gives simulation results. The paper concludes with Section 6.

2 Controller Area Network (CAN)

CAN [5, 6] is an advanced serial communication protocol for distributed real-time control systems. It is a contention-based multi-master network whose timeliness properties come from its collision resolution algorithm which gives a high schedulable utilization and guaranteed bus access latency less than $150\mu s$ for the highest-priority message on a 1 Mbit/s bus. Some other salient features of CAN are prioritized bus access, reconfiguration flexibility, and high reliability in noisy environments through CRC checks and bit stuffing.

2.1 Layered protocol

The CAN specification defines the physical and data link layers (Layers 1 and 2 in the ISO/OSI reference model). Both layers are implemented in a bus interface chip which connects the processing element (like microprocessor or smart sensor) to the bus. Such chips are available from various vendors (e.g., Intel and Philips) with a variety of features.

2.2 CAN data frame

A data message in CAN has seven fields as shown in Figure 1.

CAN allows for two message formats which can coexist on the same bus. They differ in the length of the ID field: the *standard* format has an 11-bit ID, whereas the *extended* format has a 29-bit ID. The ID field serves two purposes:

- Controls bus arbitration.
- Describes the meaning of the data (message routing).

Here we describe message routing in CAN. (Bus arbitration is described in the next subsection.) The ID, instead of containing some destination address, contains a code identifying the meaning of the data. CAN allows all or part of the ID field to be used for this purpose. For example, if the ID is 11 bits long, periodic messages from a temperature sensor may have a binary code %xxxxxx10110, where an x denotes a bit not being used for identification. All nodes desirous of knowing the current temperature will set *filters* in their bus interface chips to match the above code. Then, whenever a message with this ID code is sent on the bus, the interface chips will automatically receive it and notify the processing element of the node. This scheme is called *message filtering*.

The control field specifies the number of bytes in the data field, from 0 to 8. The CRC field contains a 15 bit CRC check, and the ack field is used to acknowledge correct reception of a message.

2.3 Bus arbitration mechanism

CAN makes use of a wired-OR (or wired-AND) bus to connect all the nodes.[1] Two logical bit representations are defined: *dominant* and *recessive*. If even a single node transmits a dominant bit on the bus, the bus will reflect a dominant bit, else it will reflect a recessive bit.

When a processor has to send a message it first calculates the message ID which may be based on the priority of the message. The ID for each message must be unique to prevent a tie.

The bus acquisition algorithm works as follows. Processors pass their messages and associated IDs to their bus interface chips. The chips wait till the bus is idle, then write the ID on the bus, one bit at a time, starting with the most significant bit. After writing each bit, each chip waits long enough for signals to propagate along the bus, then it reads the bus. If a chip had written a recessive bit but reads a dominant one, it means that another node has a message with a higher priority. If so, this node drops out of contention. In the end, there is only one winner and it can use the bus.

2.4 Application interfaces for CAN: SDS and DeviceNet

If software programs designed for CAN are to be compatible with each other, then a standard application interface will have to be defined. Currently, there are two emerging proposals: SDS from Honeywell and DeviceNet from Allen Bradley. The former is a master/slave protocol designed for simple sensors and actuators. It emphasizes a low-cost implementation. DeviceNet is a peer-to-peer connection-oriented protocol with more flexibility than SDS. Both protocols use the standard CAN message format with 11-bit IDs.

[1] In the rest of the paper we assume a wired-OR bus.

3 Workload characteristics

In computer-integrated manufacturing (CIM), the communicating devices are controllers (CPUs or PLCs), actuators (drives), and sensors. Some of these devices exchange periodic messages (such as drives) while others are more event-driven (such as smart sensors). Moreover, operators may need status information from various devices, thus generating messages which do not have timing constraints. So, we classify messages into three broad categories:

1. Hard-deadline periodic messages.
2. Hard-deadline sporadic messages.
3. Non-real-time (best effort) aperiodic messages.

3.1 Periodic messages

A common example of this type of messages is servo control of drives as in industrial cutting tools. The controller must periodically sample the current position/velocity of the drive and then send appropriate corrections to the drive. Such messages have hard deadlines, because if the update message to the drive is delayed beyond its deadline, the cutting tool may deviate significantly from its desired path, thus ruining the workpiece. In general, any system which must follow a prescribed path will have hard-deadline periodic messages (because of the periodic sensor-controller-actuator loop). Such systems include all robots and industrial cutting tools.

Note that a single periodic message will have multiple invocations, each one period apart. So, whenever we use the term *message* to refer to a periodic, we are referring to *all* invocations of that periodic.

3.2 Sporadic messages

Strictly speaking, all events in the real world are aperiodic in nature. If these events are expected to occur frequently enough, periodic monitoring can be used to detect them and take appropriate action (as in servo control).

There are other events which are not as frequent, such as temperature of a process exceeding a critical threshold. In fact, maximum interval between two such events is unbounded (event may never occur again). In such cases, using periodic messages is a waste because there is nothing to say most of the time.

Smart sensors [1] are most suitable for detecting such events. These sensors have DSP capabilities to recognize events on their own, so they signal the controller only when required. If these messages are treated as purely aperiodic, then we are assuming that they may be released at any time — even in rapid succession. If so, we will not be able to guarantee their delivery by their deadlines. Fortunately, in most real-world situations, there is a minimum interval between consecutive aperiodic events. This corresponds to a minimum interarrival time (MIT) for these messages. Such aperiodic messages which have a MIT are called *sporadic messages* [7]. Knowing the MIT of a sporadic message makes it possible to guarantee its delivery even under the worst possible situation.

3.3 Non-real-time messages

In a manufacturing environment, an operator must be able to monitor the status of every device in the system. Also, some devices (especially drives) need to communicate operational data such as torque/speed limits and diagnostic information. Such messages are non-real-time because they do not have timing constraints. Any communication protocol for manufacturing applications must be able to accommodate such messages while guaranteeing the deadlines of real-time traffic.

3.4 Low-speed vs. high-speed real-time messages

Messages in a manufacturing setting can have a wide range of deadlines. For example, messages from a controller to a high-speed drive may have deadlines of few tens of microseconds (see Section 5.1 for more details). On the other hand, messages from devices such as temperature sensors can have deadlines of a few seconds because the physical property being measured (temperature) changes very slowly. Thus, we further classify real-time messages into two classes: *high-speed* and *low-speed*, depending on the tightness of their deadlines. As will be clear in Section 4.2, the reason for this classification has to do with the number of bits required to represent the deadlines of messages.

Note that "high-speed" is a relative term — relative to the tightest deadline d_0 in the workload. So, all messages with the same order of magnitude deadlines as d_0 (or within one order of magnitude difference from d_0) can be considered to be high-speed messages. All others will be low-speed.

4 The Mixed Traffic Scheduler

As stated earlier, access to the CAN bus is controlled by the IDs of competing messages. Then the question is "how to assign IDs to messages to get the greatest possible schedulable utilization?"

To see the difficulties faced in scheduling messages on CAN, we must first consider a typical CAN bus interface chip. These chips usually have memory space for one or more messages. When a processor has to send a message, it will calculate the ID and transfer the message (with its ID) to the chip's memory. From then on, the chip will function autonomously: it will compete for the bus with the message ID, and upon getting access, it will transmit the message (there may be an option to notify the processor once a message has been sent).

Once a message has been transferred to the CAN chip for transmission, its ID will stay fixed unless the processor comes and updates it. If the ID is to be derived from the message's priority, that priority should stay fixed (at least for reasonably long periods of time). So, fixed-priority scheduling is a natural fit for these CAN chips. Each message will have a unique priority which will form its ID. This will not only uniquely identify the message for reception purposes but also schedule the message in a predictable fashion. However, in general, fixed-priority schemes give lower utilization than other schemes such as non-preemptive earliest-deadline[2] (ED). This motivates us to use ED to schedule messages on CAN. However, under ED, message priorities change dynamically and it is infeasible to continually update message IDs to reflect these priorities. This and other problems make ED impractical for CAN.

In this section we present the MTS scheduler which combines ED and fixed-priority scheduling to overcome the problems of ED. Like ED, MTS also requires that message IDs be updated but these updates can be spaced far apart in time, making MTS practical for CAN.

4.1 Fixed-priority scheduling — low utilization

As already mentioned, fixed-priority scheduling is the natural choice for currently available CAN bus interface chips. The most popular form of fixed-priority real-time scheduling is *rate monotonic* (RM) [10]. In this scheme, messages with a shorter period get higher priority than those with longer periods. RM assumes that deadline equals period, which is not always true. So, instead of RM, we can use its close relative, *deadline monotonic* (DM) scheduling [11]. With DM, messages with tighter deadlines are assigned higher priorities. Then these priorities will form the ID for each message. Once a message starts transmission, it will run to completion even if higher-priority messages are released during this time; that is, we must use the non-preemptive version of DM.

In Section 3 we saw that the network must support hard periodic as well as sporadic messages. Moreover, it must also support non-real-time traffic. DM can do this by treating sporadic messages as periodics with periods equal to their minimum interarrival times, then assigning priorities to all real-time message according to their deadlines (breaking ties arbitrarily), then using the remaining priority levels for non-real-time messages. This ensures that real-time traffic is protected from non-real-time traffic. Well-known schedulability conditions can then be used to check if all hard real-time messages are feasibly schedulable or not. We will discuss details later.

DM is a simple scheme and is easily implementable on CAN. However, to get greater schedulable utilization, we are motivated to use ED to schedule messages on CAN.

4.2 Earliest-deadline scheduling — deadline encoding problems

ED works by giving higher priority to messages with earlier deadlines-to-start-transmission at the scheduling instant. Our goal is therefore to make the IDs reflect the deadlines of messages. Moreover, each message must have a unique ID (which is a requirement of CAN). This can be done by dividing the ID into three

[2]Non-preemptive scheduling under release time constraints is NP-hard in the strong sense [8], meaning that there is no polynomial time scheduler which will always give the maximum schedulable utilization. However, Zhao and Ramamritham [9] showed that ED performs better than other simple heuristics.

priority	deadline	uniqueness

Figure 2: Structure of the ID for ED scheduling.

fields [12], as shown in Figure 2. The *deadline* field is derived from the deadline of the message. Actually, it is the logical inverse of the deadline because we want the shortest deadline to have the highest priority. To deal with the case when two messages have the same deadline, each message has a unique code which forms its *uniqueness* field. So, if two messages have the same deadline, the one with the higher uniqueness code will win. This uniqueness code also serves to identify the message for reception purposes. For ED scheduling, messages may be assigned codes arbitrarily as long as they are unique for each message [12]. However, as we will see later, the question of assigning uniqueness codes will be critical in MTS.

In Figure 2 there is also a 1-bit *priority* field which is 1 for real-time messages and 0 otherwise. This ensures that real-time messages always have higher priority than non-real-time ones.

As time progresses, deadline values get larger and larger. Eventually, they will require more bits than are available in the CAN ID field. The obvious solution is to use *slack time* [12] (time to deadline) instead of the deadline itself. But this introduces two other problems:

P1. Remaining slack time of a message changes with every clock tick. This will require IDs of all messages to be updated continually (at the start of each arbitration round). This will put too much burden on the local CPU.

P2. In Section 3 we saw that a typical communication workload in a manufacturing environment may have messages with vastly different deadlines. This means that we must encode a wide range of laxities, and there may not be enough bits in the CAN ID field to do this.

Each of the above two problems is addressed below.

Time epochs. We present a solution to **P1** in great detail. We will encounter the same problem with MTS where we will use the same solution.

One simple way to solve **P1** will be to redesign the bus interface chips to have programmable counters in appropriate positions of the ID. This way, the slack time will be updated automatically at every clock tick. However, at present such chips are not commercially available. Even if they were, they would be more expensive than chips without counters. This motivates us to investigate a software solution (a cost/performance tradeoff).

In a software solution, the CPU will still have to update the ID, but we want to reduce the frequency of these updates, i.e., spend less CPU-time on updates. Our solution uses actual deadlines (instead of slack time) but expresses them relative to a periodically increasing reference called the *start of epoch* (SOE). The time between two consecutive SOEs is called the *length of epoch*, ℓ. Then, the deadline field for message i will be the logical inverse of $d_i - \mathrm{SOE} = d_i - \left\lfloor \frac{t}{\ell} \right\rfloor \ell$, where d_i is the deadline of message i and t is the current time (it is assumed that all nodes have synchronized clocks). Value of ℓ depends on what fraction of CPU-time the designer is willing to allow for ID updates. Let this fraction be x. Let M be the MIPS of the CPU and n be the number of instructions required to do the update. Since each update must be ℓ seconds apart, $\ell = \frac{n}{xM \times 10^6}$.

So at every node, there is a periodic (timer-driven) process which wakes up every ℓ seconds and updates IDs of all ready messages according to the above equation. Comparing this with the slack-time approach, we see that if ℓ is greater than the average message length, this approach uses less CPU time. Besides, the update process is periodic and predictable, unlike the slack-time approach where the CPU must be aperiodically interrupted at the start of each arbitration round. But these benefits come at a price — our approach requires more bits for the deadline field.

Let D be the largest value of $(d-r)$ of any message, where r is the release time. Then for the slack-time approach, length of the deadline field is $m = \log_2 D$, but for our method, $m = \log_2(\ell + D)$. This is to accommodate the worst-case situation shown in Figure 3. When message i, which has the largest $(d-r)$, is released just before a new SOE, then the deadline field of message i will have value $d_i - \mathrm{SOE}_1 = \ell + D$.

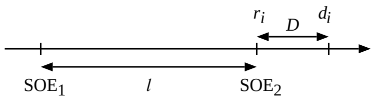

Figure 3: Largest possible value of the deadline field.

Length of ID field. The reason ED is impractical for CAN is that it requires too many bits for the deadline field in the ID for the following reason. In a typical workload, messages associated with high-speed drives may have deadlines in the hundreds of microseconds range. Other messages, such as those related to temperature sensors, may have deadlines of several seconds. If we represent deadlines at the granularity of, say, a microsecond, then more than 20 bits will be required to represent deadlines of several seconds.

One may say that if the extended format of CAN is used with its 29-bit IDs, then there will be enough bits to represent deadlines with enough left over for the uniqueness field. Unfortunately, if this scheme is used, each message will be 20 bits longer compared to the standard 11-bit format of CAN (the extended format uses two more framing bits than the standard format). This means that 20-30% bandwidth will be wasted just because of using the longer ID format.

This creates a dilemma: DM is easy to implement but it gives low utilization. We tried to use ED in hope of getting better utilization, but it increased the length of each message to the point that there was no net gain. Our solution to this problem is the MTS scheduler.

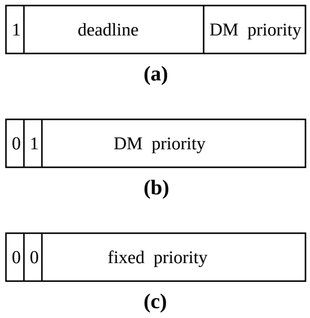

Figure 4: Structure of the ID for MTS. Parts (a) through (c) show the IDs for high-speed, low-speed, and non-real-time messages, respectively.

4.3 MTS

MTS attempts to give high utilization (like ED) while using the standard 11-bit ID format (like DM).

In DM, an 11-bit ID can represent 2048 messages. No realistic system will have this many different messages, implying that a few ID bits will remain unused. The goal is to use these bits to enhance schedulability. Since high-speed messages tend to use several times more bandwidth than low-speed ones, we will get a large improvement if we can increase the schedulability of just high-speed messages.

The idea behind MTS is to try to use ED for high-speed messages and DM for low-speed ones. First, we give high-speed messages priority over low-speed and non-real-time ones by setting the most significant bit to 1 in the ID for high-speed messages (Figure 4a). This protects high-speed messages from all other types of traffic. If the uniqueness field is to be, say 5 bits (allowing 32 high-speed messages), and the priority field is 1 bit, then the remaining 5 bits are still not enough to encode the deadlines (relative to the latest SOE). Our solution is to quantize time into *regions* and encode deadlines according to which region they fall in. To distinguish messages whose deadlines fall in the same region, we use the DM-priority of a message as its uniqueness code. This makes MTS a hierarchical scheduler. At the top level is ED: if the deadlines of two messages can be distinguished after quantization, then the one with the earlier deadline has higher priority. At the lower level is DM: if messages have deadlines in the same region, they will be scheduled by their DM priority.

We can calculate length of a region (l_r) as $l_r = \frac{\ell}{2^m - 1}$ where m is the length of the deadline field (5 in this case). This is clear from Figure 5 (shown for $m = 2$). We must reserve one coding for messages whose deadlines fall beyond the end of the current epoch.

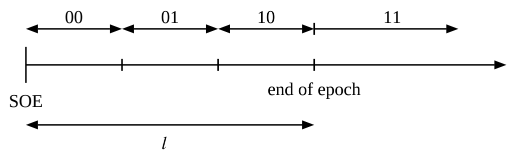

Figure 5: Quantization of deadlines (relative to start of epoch) for $m = 2$.

This leaves $2^m - 1$ codings for deadlines before the end of epoch.

Another idea may be: why not use logarithmic quantization for the deadlines? This way, we will get fine-grain discrimination for "near" deadlines and coarse-grain for "far" deadlines. This scheme would work if we were updating the deadline fields constantly, which we are not. Deadline fields of all messages are updated at the SOE and then stay fixed till the next SOE. So, what was "far" at the start of the epoch will eventually become "near" as time progresses, but its deadline encoding will stay the same coarse-grained value as it was earlier. Then, MTS will degrade to DM.

We use DM scheduling for low-speed messages and fixed-priority scheduling for non-real-time ones, with the latter being assigned priorities arbitrarily. The IDs for these messages are shown in Figures 4 (b) and (c). The second most significant bit gives low-speed messages higher priority than non-real-time ones.

This scheme allows up to 32 different high-speed messages (periodic or sporadic), 512 low-speed messages (periodic or sporadic), and 512 non-real-time messages — which should be sufficient for most applications.

4.4 Schedulability conditions

For MTS, we want off-line schedulability conditions which, when satisfied, will guarantee that all messages will meet their deadlines. We will first review such conditions for non-preemptive ED and DM, and then develop those for MTS.

Earliest-deadline: Schedulability conditions for non-preemptive ED are described by Zheng and Shin [13]. For worst-case, sporadic messages are treated as periodic with period equal to their MIT. Then, if all messages are released at the same time $t = 0$ (creating the worst-possible congestion), they will still be schedulable if the following two conditions hold:

1. $\sum_{j=1}^{n} C_j/T_j \leq 1$.
2. $\forall t \in S$, $\sum_{i=1}^{n} \lceil (t - d_i)/T_i \rceil^+ C_i + C_p \leq t$, where $S = \cup_{i=1}^{n} S_i$, $S_i = \{d_i + nT_i : n = 0, 1, \cdots, \lfloor (t_{max} - d_i)/T_i \rfloor\}$, and $t_{max} = \max\{d_1, \cdots, d_n, (C_p + \sum_{i=1}^{n}(1 - d_i/T_i)C_i)/(1 - \sum_{i=1}^{n} C_i/T_i)\}$.

where T_i, C_i, d_i are the period, length, and deadline of message i, C_p is the length of the longest possible packet, and $\lceil x \rceil^+ = n$ if $n - 1 \leq x < n$, $n = 1, 2, ...$, and $\lceil x \rceil^+ = 0$ for $x < 0$.

The first condition ensures that maximum utilization does not exceed capacity, and the second one ensures that each message with deadline $\leq t$ can finish by t.

We must extend the above conditions for the situation when some messages have phase offsets and cannot all be released at the same time. A common example of such messages are those involved in servo control. The drive first sends a feedback to the controller which then sends a command back to the drive. These two messages can never be released at the same time (because one synchronizes off the other). The worst-case occurs when one is released at $t = 0$ and the other at its phase offset. We can model every such pair of messages by letting the first message have phase $\phi_1 = 0$ and the second message with $\phi_2 =$ its phase offset *relative* to the first message. Then the above conditions become:

1. $\sum_{j=1}^{n} C_j/T_j \leq 1$.
2. $\forall t \in S$, $\sum_{i=1}^{n} \lceil (t - d_i - \phi_i)/T_i \rceil^+ C_i + C_p \leq t$, where $S = \cup_{i=1}^{n} S_i$, $S_i = \{d_i + nT_i : n = 0, 1, \cdots, \lfloor (t_{max} - d_i - \phi_i)/T_i \rfloor\}$, and $t_{max} = \max\{d_1, \cdots, d_n, (C_p + \sum_{i=1}^{n}(1 - d_i/T_i)C_i)/(1 - \sum_{i=1}^{n} C_i/T_i)\}$.

Deadline monotonic: For the non-preemptive case, a message i is feasible if all higher-priority messages are feasible and i finds an opportunity to start transmission sometime during $[0, d_i - C_i]$. So, all we have to do is look at messages with priority higher than that of i and assume that they are the only messages in the system. If the bus ever becomes idle during $[0, d_i - C_i]$, then i is schedulable. If messages are numbered according to their priority with $j = 0$ being the highest-priority message, then i is schedulable if [14]:

$\exists t \in S$, $\sum_{j=1}^{i-1} \lceil (t - \phi_j)/T_j \rceil C_j + C_p \leq t$, where $S =$ {set of all release times of messages $0, 1, \cdots, i-1$ through time $d_i - C_i$} $\cup \{d_i - C_i\}$, and ϕ_j are the *relative* phase offsets.

To check schedulability of a set of messages, repeat the above check for each message.

MTS: First, we will discuss the schedulability check for high-speed messages and then look at low-speed ones. The worst-case loading conditions for high-speed messages result when there is

1. worst possible traffic congestion, and
2. worst possible deadline encoding.

The first situation is created by releasing all messages at the same time t_0 (messages with non-zero ϕ_i will be released later). To see the second situation, consider Figure 6. Suppose we want to determine worst-case schedulability of message i. We want to maximize the number of messages which get priority over i. If deadline-to-start of i falls at the start of a region, then all messages with earlier deadlines-to-start

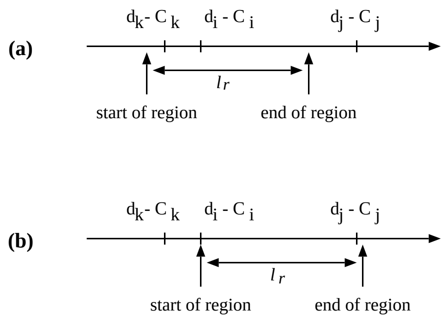

Figure 6: Suppose j has higher DM-priority than i but k does not. Then in (a), i has the highest priority, whereas in (b), it has the lowest. So (b) is the worst-case situation for i.

will fall in earlier regions and get higher priority. Moreover, it will allow the maximum number of messages — whose deadlines-to-start are greater than that of i — to fall in the same region as i. If they have higher DM-priority, they will also go before i.

Now, we can draw a parallel between schedulability conditions for MTS and those for DM. To determine schedulability of some message, consider its first invocation i and all j with priority greater than that of i under worst-case situations (for MTS, this means deadline-to-start of i coincides with the start of a region). Then, from the above discussion, message invocations j will have priority over i if:

1. $(d_i - C_i) > (d_j - C_j)$, or
2. (a) $(d_i - C_i) < (d_j - C_j) \leq (d_i - C_i + l_r)$, and
   (b) DM priority of j is greater than that of i, and
   (c) j is released before $d_i - C_i$.

Note that j represents individual invocations, not entire "messages."

Then, if we consider *only* those invocations j which satisfy the above conditions and schedule them according to MTS and the bus ever becomes idle during interval $[t_0, t_0 + (d_i - C_i)]$, then message i will get a chance to run and it will be schedulable. This check must be repeated for each high-speed message in the workload. Formally, a message is schedulable if its first invocation i satisfies the condition:

$\exists t \in S, \;\; \sum_{\forall j} \lceil \frac{t-(t_0+\phi_j)}{T_j} \rceil C_j + C_p \leq t,$
where each j must satisfy the above conditions, S ={set of release times of each j} $\cup \{d_i - C_i\}$, C_p is the size of a longest possible packet, and ϕ_j are the relative phase offsets.

Checking schedulability of low-speed messages is simple — just check DM schedulability for each low-speed message. Since high-speed messages have shorter deadlines than low-speed ones, they will automatically have higher DM priority (which is exactly what we want).

5 Evaluation

We have designed MTS to offer better performance and schedulability than DM. Since deadlines in MTS are quantized, we would expect the performance of MTS to be close to that of ED if somehow message length did not increase when using ED. So, let ED* be an ideal (imaginary) scheduling policy which works the same as ED but requires only an 11-bit ID. Then ED*'s performance should be an upper bound on MTS's.

In the absence of any analytical results to prove which scheduler is better, we have to resort to simulations to evaluate performance. For this, we need a workload. Rather than randomly generating messages, we came up with a single realistic workload model — realistic in the sense that all messages are associated with devices commonly used in manufacturing such as drives, sensors, etc.

In this section, we will first describe our workload model, then present simulations to evaluate MTS's performance as compared to DM and ED*.

5.1 Workload model

In computer-integrated manufacturing (CIM), high-speed messages are usually related to servo control of drives. Currently-available drives can operate at speeds of up to 1200 rpm or higher. Frequency of updates needed to control such drives is a function of how much error (e.g., in position and velocity) is tolerable for the application at hand [15], but these frequencies can be as high as 10 kHz. As already mentioned, a pair of messages are involved in controlling each drive. First, the drive will send feedback to the controller. The controller will require some time to compute the new command, then send this to the drive. The drive will adjust according to the new command, and after some delay, repeat the cycle. For simulation purposes, we assume that the delay on each side is 10% of the cycle time. So if the cycle is of length c, the cycle will start by the drive sending a message at time t_0 whose deadline will be $t_0 + 0.4c$. At time $t_0 + 0.5c$, the controller will send the new command to the drive with deadline $t_0 + 0.9c$, and then the cycle will repeat forever.

An example of high-speed sporadic messages will be those associated with a robotic gripper. Each finger in the gripper will have its own drive. Since the finger must be controlled precisely, the update frequency for these drives will be quite high. Now, to minimize the time required to grasp an object, velocity control may initially be used to close the fingers rapidly. When the fingers touch the object, the controller must switch to force control to avoid both slippage at one hand and damage to the object (from excessive force) at the other. To detect first-contact, each finger is equipped with a sensor which will send a message on contact [16]. Since the controller must immediately switch control strategies, this message will not only have a hard deadline, but also a very tight one. For simulation, we assume that this message must be received by the controller within one-fourth of the cycle time of the drive to give the controller enough time to switch control strategies. Also, the minimum interval between a robot picking up two separate objects will be at least several seconds, which is the minimum interarrival time (MIT) for these sporadic messages.

Low-speed periodic messages can be associated with several types of devices such as slow drives (e.g., 200 rpm) and periodic polling of dumb sensors. Such messages will have periods and deadlines in the order of several milliseconds.

Low-speed sporadic messages may result, for example, from a smart sensor monitoring the temperature of a process. If the temperature exceeds a certain limit, the sensor transmits a warning to some controller which may shut down the process. Since temperature changes slowly, such messages will have deadlines of tens of milliseconds or more. Once the process is stopped, it will take at least several minutes to restart it, which will be the minimum interarrival time for these messages.

Having a qualitative idea of the types of messages in CIM, we can now choose specific values for message periods and deadlines to be used in simulation.

Consider a drilling machine with an attached robot arm to move workpieces. Suppose the robot has a two-fingered gripper. Since the fingers must be controlled precisely, their drives will have a high update frequency which we chose to be 8 kHz. The robot arm will have several joints, each with a drive. They require a high degree of precision as well, but not quite as high as the fingers, so we chose 6 kHz as their update frequency. One drive is needed to move the drill along a single axis (in and out of the workpiece). This carriage requires even less precision, so we chose its frequency to be 4 kHz. Finally, the drill itself has a drive whose speed is limited by the maximum possible metal removal rates. We chose its update frequency to be 2 kHz. Each of these drives has a pair of messages associated with it whose deadlines are assigned according to the general framework presented earlier.

For each finger there is a smart contact sensor. This means one sporadic message per finger with a deadline of one-fourth of the drive cycle (30μs).

For low-speed messages, we arbitrarily chose periodic messages to have periods of 20ms and deadlines of 8ms, and sporadic messages with deadlines of 5ms and MITs of 5s.

Table 1 summarizes the choice of parameters. It also gives the number of messages of each type for our simulations. The values marked with "$\star$" are default and will be varied during simulation.

Now, we must consider the length of each message. The standard CAN message format requires 47 framing bits per message for ID, CRC, etc. For position or velocity control of a drive, 32-bit command and 32-bit feedback variables are required [17], making each message 79 bits long. We use this length for each periodic message in the workload as well as for the longest possible message (C_p).

Aperiodic messages are used to notify a controller of some event. A unique message ID is enough for the controller to recognize which event is being signaled, so

High-speed messages				
Type	*Class*	*Period/MIT*	*Deadline*	*# of mssg.*
Fingers	Periodic	125.0μs (8 kHz)	50.0μs	4
Joints	Periodic	166.7μs (6 kHz)	66.6μs*	6*
Carriage	Periodic	250.0μs (4 kHz)	100.0μs	2
Drill	Periodic	500.0μs (2 kHz)	200.0μs	2
Sensors	Aperiodic	2s	30.0μs*	2*

Low-speed messages			
Class	*Period* (ms)	*Deadline* (ms)	*MIT*
Periodic	20.0	8.0	—
Aperiodic	—	5.0	5s

Table 1: Simulation workload.

Figure 7: Schedulability under DM, MTS, and ED* as number of 6 kHz periodic messages are varied.

aperiodics can have 0 data bytes, making them 47 bits long each. This is the length we use in our simulations.

The last parameter to consider is the length of epoch (ℓ) for the simulations. We assume a 20 MIPS CPU. If each deadline update requires 1000 instructions and we allow 5% of CPU time for these updates, then $\ell = \frac{1000}{(0.05)(20\times10^6)} = $ 1ms. Then each region has length $l_r = \ell/(2^m - 1) = $ 1ms$/(2^5 - 1) = 32.3\mu$s.

5.2 Simulation results

Current CAN chips are designed for a 1 Mb/s physical medium. This speed is insufficient for applications like high-speed servoing. So in our simulations, we assume a 10 Mb/s physical medium (which implies a time granularity of 0.1μs).

In all our simulations, we use the general drilling machine workload described in the previous subsection. Using the schedulability conditions of section 4.4, we want to check schedulability of workloads under different scheduling policies when one workload parameter (such as the number of a certain type of message) is varied. We expect that MTS will perform better than DM and close to ED*. Initially, we evaluate MTS's performance for high-speed messages. Then we go on to low-speed messages.

Varying number of high-speed periodics: We varied the number of 6 kHz periodic messages. This

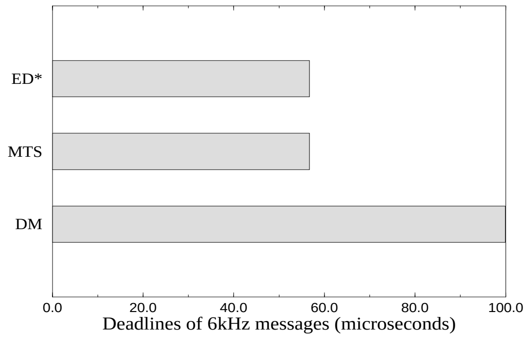

Figure 8: Deadlines of 6 kHz periodic messages for which workload is *unschedulable* under DM, MTS, and ED*.

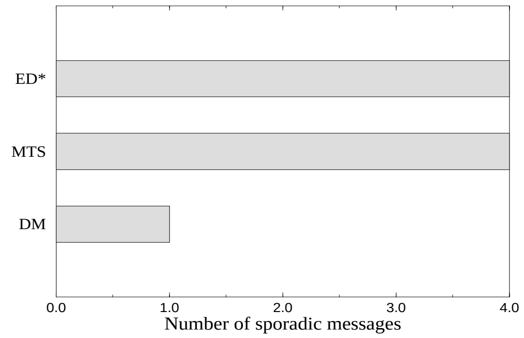

Figure 9: Schedulability under DM, MTS, and ED* as number of sporadic messages are varied.

can be thought of as varying the number of joints in the robot arm.

All other high-speed message types have fixed parameters as shown in Table 1: two fingers (four periodics) and two contact sensors (two sporadics); one drill carriage (two periodics); and one drill (two periodics). No low-speed messages are needed at this point since they do not affect the schedulability of high-speed messages.

The schedulability results for DM, ED*, and MTS are shown in Figure 7. DM can handle only five 6 kHz messages (total utilization of 58.5%), whereas both ED* and MTS can handle up to 8 each (72.7% utilization). At least for this workload, MTS performed better than DM and same as ED*.

Varying deadlines of high-speed periodics: In this simulation, we varied the deadlines of 6 kHz periodic messages while keeping their number fixed at 6. Parameters for all other message types were the same as in Table 1, giving a utilization of 63.2%.

This workload was found unschedulable under DM at the deadlines of 99.9μs or less (Figure 8). On the other hand, the workload was schedulable under both MTS and ED* even at the smallest possible deadline of 56.8μs. (If deadlines are decreased to 56.7μs, then the message set is unschedulable because C_p + two 8 kHz messages + three 6 kHz messages + two sporadic messages need at least 56.8μs to complete).

Again, MTS performed much better than DM and at the same level as ED*.

Varying number of high-speed sporadics: In this simulation, we vary the number of sporadic messages while keeping their deadlines fixed at the value shown in Table 1. The number and deadlines of all periodic messages are fixed at the values in Table 1, giving a utilization of 63.2%.

Our simulations showed that DM can handle only one sporadic message in the workload, whereas both MTS and ED* handled up to four (Figure 9).

DM-schedulability is shown to decrease rapidly as the number of sporadic messages increase. On the other hand, MTS (as well as ED*) is capable of handling a significant number of sporadic messages in the workload.

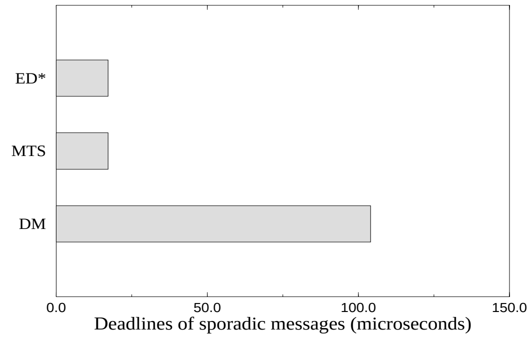

Figure 10: Deadlines of sporadic messages for which workload is *unschedulable* under DM, MTS, and ED*. Number of 6 kHz periodics is 6 for this simulation.

Varying deadlines of high-speed sporadics: In this simulation, we vary the deadlines of all high-speed sporadic messages while keeping all other parameters at their default values.

The workload becomes unschedulable under DM for sporadic message deadlines of 104.1μs or lower (Figure 10), but continues to be feasible under both MTS and ED* till the deadline of 17.3μs, which is minimum possible (C_p + two sporadic messages take at least 17.3μs to complete).

To better compare MTS against ED*, we tested the two under heavier loads. We increased the number of 6 kHz messages to 10 (82.2% utilization), then started varying sporadic message deadlines. This workload was unschedulable under MTS for deadlines of 151.5μs or less, whereas it became unschedulable under ED* only at deadlines of 72.5μs or less (Figure 11). So under heavy loads and tight deadlines, ED* outperforms MTS, as expected.

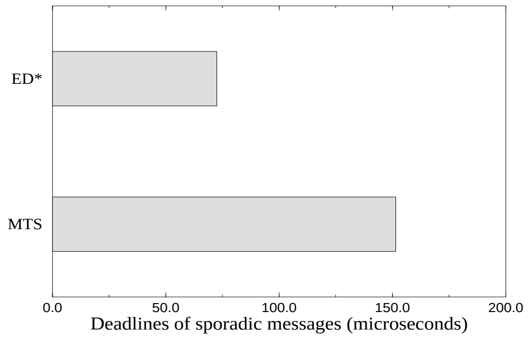

Figure 11: Deadlines of sporadic messages for which workload is *unschedulable* under MTS and ED*. Number of 6 kHz periodics is 10 for this simulation.

Low-speed messages: Under default loading conditions, high-speed messages use up 63.2% of the bandwidth. The remaining 36.8% can accommodate several hundred low-speed sporadic and periodic messages. Even 10% of the bandwidth is enough to accommodate about a hundred or so low-speed messages. Thus, the schedulability of low-speed messages is not a problem. Our simulations showed no real difference between DM, MTS, or ED* in scheduling low-speed messages for a fixed load of high-speed messages.

6 Conclusion

The two most attractive features of CAN are a short worst-case bus access latency and a bus acquisition scheme based on the priority of messages, both of which give CAN the potential for high performance and fewer missed deadlines in distributed control systems. However, the bus arbitration mechanism must be used properly with careful design of the message ID; otherwise, CAN will give low utilization.

In this paper we presented the MTS scheduler which allowed three different types of messages — hard sporadic, hard periodic, and non-real-time aperiodic — to be carried on the same bus. Then, we designed the message ID which implements MTS on CAN. MTS not only gives high schedulability but is also easily implementable on CAN.

Through simulations we compared MTS with DM and ED* (an idealized form of ED). We used a realistic workload for these simulations, modeling drives, controllers, and sensors typically found in CIM systems. As expected, MTS performed much better than DM and only slightly worse than ED*, and in many cases, it matched the performance of ED*.

References

[1] J. Brignell and N. White, *Intelligent Sensor Systems*, Bristol, Philadelphia, 1994.

[2] A. Tanenbaum, *Computer Networks*, Prentice-Hall, Englewood Cliffs, N.J., 1989.

[3] H. Kopetz and G. Grunsteidl, "TTP – a protocol for fault-tolerant real-time systems," *IEEE Computer*, vol. 27, no. 1, pp. 14–23, January 1994.

[4] J. K. Strosnider and T. E. Marchok, "Responsive, deterministic IEEE 802.5 token ring scheduling," *Journal of Real-Time Systems*, vol. 1, no. 2, pp. 133–158, September 1989.

[5] *CAN Specification Version 2.0*, Robert Bosch GmbH, 1991.

[6] *SAE Handbook*, Society of Automotive Engineers, pp. 20.379–20.392, 1993.

[7] A. K. Mok, "Fundamental design problems of distributed systems for the hard real-time environment," *Ph.D thesis*, MIT, 1983.

[8] K. Jeffay, D. F. Stanat, and C. U. Martel, "On non-preemptive scheduling of periodic and sporadic tasks," in *Proc. Real-Time Systems Symposium*, pp. 129–139, 1991.

[9] W. Zhao and K. Ramamritham, "Simple and integrated heuristic algorithms for scheduling tasks with time and resource constraints," *Jounal of Systems and Software*, vol. 7, pp. 195–205, 1987.

[10] C. L. Liu and J. W. Layland, "Scheduling algorithms for multiprogramming in a hard real-time environment," *Journal of the ACM*, vol. 20, no. 1, pp. 46–61, January 1973.

[11] J. Y.-T. Leung and J. Whitehead, "On the complexity of fixed-priority scheduling of periodic, real-time tasks," *Performance Evaluation*, vol. 2, no. 4, pp. 237–250, December 1982.

[12] K. G. Shin, "Real-time communications in a computer-controlled workcell," *IEEE Trans. Robotics and Automation*, vol. 7, no. 1, pp. 105–113, February 1991.

[13] Q. Zheng and K. G. Shin, "On the ability of establishing real-time channels in point-to-point packet-switched networks," *IEEE Trans. Communications*, pp. 1096–1105, February/March/April 1994.

[14] D. D. Kandlur, K. G. Shin, and D. Ferrari, "Real-time communication in multihop networks," *IEEE Trans. Parallel and Distributed Systems*, vol. 5, no. 10, pp. 1044–1056, October 1994.

[15] M. Ehsani, I. Hussain, and K. R. Ramani, "An analysis of the error in indirect rotor position sensing of switched reluctance motors," in *International Conference on Industrial Electronics, Control and Instrumentation*, pp. 295–300, October 1991.

[16] S. Soloman, *Sensors and Control Systems in Manufacturing*, McGraw-Hill, Inc., New York, 1994.

[17] *Electrical Equipment of Industrial Machines — Serial Data Link for Real-Time Communication between Controls and Drives*, International Electrotechnical Commission, Rev. 8, 1994.