A Causal Message Ordering Scheme for Distributed Embedded Real-Time Systems*

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

In any distributed system, messages must be ordered according to their cause-and-effect relation to ensure correct behavior of the system. Causal ordering is also essential for services like atomic multicast and replication. In distributed real-time systems, not only must proper causal ordering be ensured, but message deadlines must be met as well. Previous algorithms which ensure such behavior include the Δ*-protocol family [1] and the MARS approach [2]. However, both these algorithms give large response times by delaying all messages for a fixed period of time. In this paper we show that for small- to medium-sized real-time systems (consisting of a few tens of nodes) as are commonly used for embedded applications, it becomes feasible to extend the* Δ*-protocol so that instead of delaying all messages for a fixed period, each message is delayed according to its deadline. Our algorithm requires certain message deadlines to be adjusted by the application designer, and we show that for small-scale applications such as those used in embedded systems, this adjustment is feasible and can be automated by the use of proper CAD tools.*

1. Introduction

Embedded real-time systems are today being used in a wide range of applications, from automotive and robotics to factory automation and process control. The physically and logically distributed nature of such applications has prompted researchers to look for distributed solutions. Distributed solutions have the potential for enhanced performance as well as modularity and a high degree of fault-tolerance, but designers are faced with a problem which was a non-issue in centralized systems: ensuring that all nodes have the correct and consistent view of the environment.

In a typical distributed real-time system, the task of monitoring and controlling various aspects of the environment is divided among the nodes. When a node observes some event, it informs other relevant nodes of this event through messages. Due to unpredictable communication delays, different nodes may learn of different events at different points in time, causing some nodes to have an incorrect view of the environment. They will act inconsistently and perhaps cause damage to life and property. For example, suppose a smart temperature sensor sends two updates regarding the external temperature. If these messages get reordered during transmission, the receiving nodes will perceive the old temperature reading to be the correct one even though it is outdated. So for correct and reliable operation of a distributed real-time system, messages must be ordered so that their cause-and-effect relationship is preserved. This ordering must be performed by the operating system to reduce the burden on the application programmer. That way, the applications can correctly assume that events occur in the order they observe, allowing application designers to write software the way they are used to in centralized systems. This ordering primitive can then be used as a basis for higher-level fault-tolerance mechanisms such as atomic multicast [3] and replication for real-time systems.

The problem of ordering events in non-real-time systems has been the focus of research for more than a decade [4, 5], but has only recently been investigated for real-time systems [2, 6–9]. Real-time systems differ from non-real-time systems in that one node may affect other nodes through the *external* environment, which is why causal ordering schemes for non-real-time systems do not work for real-time sys-

*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.

tems [10, 15]. Causal ordering schemes for real-time systems fall into two broad categories [9]: *clock-driven* and *timer-driven*. The former makes use of a global time base whereas the latter relies on local timers. Timer-driven protocols have a shortcoming that they can order only those events which have a large time-separation (tens of milliseconds)[1]. This is not acceptable for our target applications, so we will not consider timer-driven protocols any further.

Clock-driven protocols can order events with much smaller separations (comparable to the precision of the global clock). They make use of synchronized clocks to delay messages in one way or the other to ensure that the order of events seen by the application is indeed the correct order. Such schemes include the Δ-protocol family [1] and the MARS approach [2]. Each has a simple criterion for delaying messages but the price of this simplicity is slower speed. Faster response can be obtained only by increasing network bandwidth and processor speed. In large and complex real-time systems (consisting of hundreds of nodes or more), this simplicity may be enough to justify the extra cost of faster networks and processors, but not in small- to medium-sized embedded systems. Such systems usually consist of a few tens of nodes interconnected by a real-time LAN. They are typically produced in large volumes (tens of thousands or sometimes even millions of units) for applications like automotive control and factory automation. These large production volumes make it imperative that production costs be kept to a minimum because extra costs of even a few dollars per unit translate into an overall loss of millions of dollars. For such small- to medium-sized applications we need an ordering scheme which performs correctly and efficiently without delaying all messages unnecessarily thus giving faster responses with cheaper networks and processors. In this paper we present an extension to the Δ-protocol which, instead of delaying all messages for a fixed period Δ, delays each message according to its deadline. We show that implementing this scheme for our target applications is feasible and the benefit is faster response and/or cheaper networks and processors than those provided by the original Δ-protocol or the MARS approach.

The drawback of our scheme is that it requires deadlines of causally-related messages to be adjusted by the application designer in a specific manner. Then, the all-important question is whether such an adjustment is feasible or is it too much of a burden for the application designer. We show that these deadline adjustments can be automated by CAD tools as long as the scale of the target application is not too large and enough *a priori* information is available (as is commonly the case for embedded systems). This way, the overhead of ordering messages shifts from run-time to design-time. The advantage is that at design-time, CAD tools can be used to facilitate the design process and reduce overhead.

[1] In the terminology of [11], events must be separated by at least σ, the *steadiness* of the protocol, which is usually tens of milliseconds.

In the next section we give some definitions and review the existing ordering schemes for real-time systems and identify their inefficiencies. Then, in Section 3 we present our ordering scheme, identify its design-time requirements, and show how a CAD tool can help in fulfilling these requirements. In Section 4, we use simulations to show that the run-time impact on network schedulability of our ordering scheme is almost negligible, and we conclude with Section 5.

2. Previous Work

A typical distributed real-time system consists of a set of nodes interconnected by an arbitrary topology real-time communication network such as Controller Area Network (CAN) [12] and token bus [13]. In such systems, nodes usually inform each other of events by exchanging messages. Since the communication subsystem may take a variable amount of time to deliver messages, there is always the potential for messages being reordered en route to their destinations. Even in LANs, arbitration delays and buffering on end-nodes may cause messages to be reordered. So if nodes have to rely solely on the receive order of messages to order events, then they are likely to act inconsistently.

Before proceeding further, let us define some terms. Sending a message m, called the *send* event and denoted by `send(`m`)`, occurs when a process P passes m to the communication subsystem on its node. Note that actual transmission of m on the network may occur later due to arbitration delays. Each message has a destination process denoted by `dest(`m`)`. When the communication subsystem on the node on which `dest(`m`)` runs gets m from the network, this is called the *receive* event. The OS may hold m for some time, then deliver it to the application, which is known as the *deliver* event denoted `deliver(`m`)`.

To tackle the problem of consistent and correct ordering of events, a set of possible relationships between messages in a distributed system have been identified. The most commonly-known relationship is *logical order* [10, 14]:

Logical Order: Logical order is usually defined based on Lamport's *happens-before* relationship [4]. Event E_1 is said to happen-before event E_2 (denoted as $E_1 \rightarrow^l E_2$) if:

- E_1 and E_2 are events on the same node and E_1 occurs before E_2, or
- E_1 is a *send* event and E_2 is a corresponding *deliver* event, or
- $E_1 \rightarrow^l E_i$ and $E_i \rightarrow^l E_2$ for some event E_i.

Preserving logical order of events is enough to ensure correctness in non-real-time systems [4, 5], but not in real-time systems because of "clandestine" communication which can occur through what are known as *hidden* channels [2, 6, 10, 15]. A simple example of such communication is a process P on one node taking an action (through an actuator) and notifying other processes of it through messages. Sensors on other nodes may detect a change in the environment due to this action before those nodes receive the message from P, leading processes on these nodes to act inconsistently. This type of communication from an actuator to a sensor can be thought of as a "virtual" message. In real-time systems, such virtual messages must be considered as well — in addition to the "real" messages — when properly ordering events and messages. If we extend the definitions of *send* and *deliver* to apply not only to real messages but to virtual ones as well [11] (in the above example, the *send* event will be the actuation and the *deliver* event will be its detection by a sensor), then we can define the *causal order* relationship between events:

Causal Order: Causal order is defined based on the *precedence* relationship between events. Event E_1 is said to precede event E_2 (denoted as $E_1 \rightarrow E_2$) if:

- E_1 and E_2 are events on the same node and E_1 occurs before E_2, or
- E_1 is a *send* event and E_2 is a corresponding *deliver* event (for real or virtual message), or
- $E_1 \rightarrow E_i$ and $E_i \rightarrow E_2$ for some event E_i.

Our objective is to ensure that messages are delivered in accordance with their causal order. This is known as *causal delivery* and is defined as follows [11]:

Causal Delivery: If $m_j \rightarrow m_i$ and $\texttt{dest}(m_j) \equiv \texttt{dest}(m_i)$, then causal delivery ensures $\texttt{deliver}(m_j) \rightarrow \texttt{deliver}(m_i)$, where m_i, m_j can be real or virtual messages.

We have already pointed out that ordering schemes which preserve only logical order cannot provide causal delivery. To secure causal order (actually, *potential* causal order), we can make use of the *temporal order* relationship between messages:

Temporal Order: The temporal order of events in a distributed system is defined in terms of an omniscient external observer. This observer possesses a single reference clock and timestamps each event by this clock. So event E_1 temporally precedes event E_2 if timestamp(E_1) < timestamp(E_2). In simpler terms, if E_1 occurs before E_2 in real-time, then E_1 temporally precedes E_2.

Two well-known clock-based schemes which use temporal order to correctly order events in real-time systems are described next. They both depend on the observation that "temporal order is a prerequisite for causal order" [2]. In other words, if event E_1 was the cause of event E_2, then E_1 must temporally precede E_2.

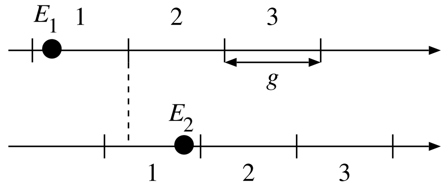

Figure 1. Causally-related events must be separated by at least $2g$, otherwise they may get the same timestamp.

2.1. The Δ-Protocols

Under the Δ-protocols [1], each message m is timestamped at the source with the time t of the $\texttt{send}(m)$ event. The OS at the destination node holds m until time $t + \Delta$ before delivering it to the application. Δ is a system constant and is made greater than the sum of the maximum message transmission delay and clock precision. Since all messages take time Δ to be delivered, the temporal order is preserved and causal order results automatically.

There are some limitations of all clock-driven ordering protocols (including the Δ-protocols) as described in [6] and elsewhere, because real clocks have non-zero clock granularity[2] g and tend to drift apart from each other. This limits the system's ability to order events timestamped on different nodes. If two clocks can be apart by as much as g, then two events must be separated by at least $2g$ in real-time for them to be reliably ordered by the system as illustrated by Figure 1. If events E_1 and E_2 are separated by less than $2g$, it is possible for them to get the same timestamp. This means that if the two events really have a cause-and-effect relation and the minimum separation between them is δ, then for correct operation, the clock synchronization must be tight enough to satisfy $2g \leq \delta$.

2.2. The MARS Approach

The MARS approach to causally ordering messages in real-time systems is similar in spirit to the Δ-protocol in that all messages are delayed for a (more or less) fixed duration. The MARS approach delays messages at both the source as well as the destination. It is based on the sparse timebase concept [2]. The system proceeds in lock-step and each step

[2]For granularity g to be meaningful, it must be larger than the precision of the clock [9].

is one clock tick of usually large granularity (in the order of a millisecond [2]). All message send and deliver events are constrained to occur at the clock ticks and the clock granularity is larger than maximum message transmission time. So all messages sent at clock tick k will be delivered at clock tick $k+1$. As long as δ is greater than the clock granularity, causal order is ensured. □

The main limitation of the above schemes is that each message gets delayed a period (say Δ) which is greater than the longest message transmission time. In real-time systems, each message has an associated deadline by which it must reach its destination. If these deadlines are to be met, Δ must be made smaller than the tightest message deadline. This in effect forces each message to have the same tight deadline which adversely affects network schedulability (schedulable utilization). As a result, the application designer must either tolerate slower responses or switch to faster networks and processors. Neither of these alternatives are acceptable in cost-conscious embedded systems where fast response is desired at low costs. In the next section we present a modification to the Δ-protocol which delays messages according to their individual deadlines instead of a fixed delay Δ for all messages. This requires deadlines of causally-related messages to be adjusted in a specific manner. The important question is then: is such an adjustment feasible or not? We show that for our target applications, adjusting message deadlines is feasible and can be automated with a CAD tool currently under development in the Real-Time Computing Laboratory at the University of Michigan.

3. Causal Ordering in Embedded Systems

The simplicity of the schemes described in the previous section is very appealing and very useful in designing large, complex real-time systems. However, their overhead is unacceptable in smaller, mass-produced embedded systems. Fortunately, the small scale of these systems presents opportunities for optimization which are not feasible in larger systems. Here we present a new causal ordering scheme which uses knowledge of message deadlines to give better response times than other known ordering schemes for real-time systems.

3.1. Message Deadlines

Messages in real-time systems have some deadlines imposed on them by the response time requirements of the application which in turn are usually dictated by the external environment being controlled by this real-time system. Let's call this the *environmental restriction* on message deadlines. But when two messages m_i and m_j are causally related and have the same destination, there is an additional restriction on their deadlines:

Causal restriction: If $m_j \rightarrow m_i$ and $\texttt{dest}(m_j) \equiv \texttt{dest}(m_i)$, then D_j must be $\leq D_i$,

where D_k is the absolute deadline of m_k. If d_k is the relative deadline of m_k and m_k is sent at time t_k, then $D_k = t_k + d_k$. The reason for this restriction is obvious: if the OS must deliver m_j before m_i without violating the environment-imposed deadline of m_i, then m_j must be received before m_i's deadline D_i expires.

Notice that $D_j < D_i$ is easy to satisfy. Both the Δ-protocol and the MARS approach satisfy this by forcing all messages to have the same relative deadline (say Δ). If the environmental restriction is to be satisfied as well for all messages, then we need $\Delta \leq$ tightest environment-imposed deadline. This requires faster processors and networks to be used than would be needed otherwise.

The small scale of our target applications provides an alternative. For such small applications, it becomes feasible for the application designer to identify causal relationships between messages and then adjust the message deadlines so that they satisfy $D_j = D_i$. Next, we describe our causal ordering scheme, then, in its context, we discuss details (in Section 3.3) of how the causal restriction can be easily satisfied.

3.2. Causal Ordering Scheme

Initially, assume perfectly synchronized-clocks with a clock granularity g close to zero (we will relax these assumptions later). Then, our scheme works as follows.

1. The OS at each node knows the relative deadlines of all messages it is to receive from all other nodes. There are several ways of achieving this, including:
   - Deadline values can be statically programmed into each node.
   - The transmitting nodes can inform all receiving nodes of message deadlines during a startup phase.
2. The OS at the transmitting node timestamps each message m_k with the time of the $\texttt{send}(m_k)$ event. Let t_k be the timestamp of m_k.
3. The OS at the receiving node will hold message m_k until time $D_k = t_k + d_k$, then deliver m_k to the application. If more than one message have the same D_k, they will be ordered by their t_k.

This scheme will work correctly as long as the application designer ensures the causal restriction on message deadlines is satisfied. Then, any message m_j such that $m_j \rightarrow m_i$

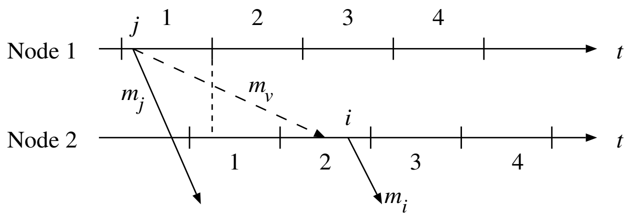

Figure 2. Uncertainty due to non-ideal clocks. Broken line indicates a virtual message.

will have $D_j \leq D_i$ as well as $t_j < t_i$ (event j must occur before event i if j is to causally precede i [2]). Network scheduling [16–18] will ensure that both m_j and m_i meet their deadlines so that m_j will be sure to be received at the destination node before time D_i. The OS will order these messages by their D_k (and t_k in case of ties) so that m_j will be delivered before m_i.

Now, we modify this scheme to work with realistic clock synchronization with non-zero granularity g. Clocks on two nodes can be out of sync by any value less than g. Known clock synchronization schemes for LANs can ensure g to be as small as 10–20 μs [19, 20]. As with other clock-based ordering schemes (Section 2), two events must be separated by at least $2g$ for the system to be able to order them properly.

A non-zero g combined with virtual messages creates the problem shown in Figure 2. Suppose the minimum separation between events i and j is 2 clock ticks by virtue of a virtual message m_v. Then, by the causal restriction, d_j must be $\leq d_i + 2$ if D_j is to be $\leq D_i$. Let $d_j = d_i + 2$. But since the clocks are not perfectly synchronized, it can happen that $t_j = t_i - 1$ (instead of $t_i - 2$). Then $D_j = t_j + d_j = t_i + d_i + 1 = D_i + 1$, which may cause m_j to be delivered after m_i. So with non-zero g and virtual messages, the protocol as described above can no longer correctly order events separated by $2g$. Note that if we replace m_v by a real message m_l with $d_l = 2$, then it will not be delivered till time $t_l + d_l = 1 + 2 = 3$ by node 2's clock. Then t_i will be $t_j + 2$ and there will be no problems. So the problem occurs because virtual messages, unlike real ones, cannot be delayed at the destination.

The above example shows that the causal restriction as previously stated is no longer enough for non-ideal clocks with virtual messages and must be modified as follows (m_p is the immediate precedent[3] of the generic message m_k):

Causal restriction (non-ideal clocks): If $m_j \rightarrow m_i$ and `dest(`m_j`)` $\equiv$ `dest(`m_i`)`, then D_j must be $\leq D_i - g$, where $D_k = \tau_k + d_k$ and

$$\tau_k = \begin{cases} \tau_p + d_p & m_p \text{ is real or } m_k \text{ is virtual} \\ \tau_p + d_p - g & m_k \text{ is } m_i \text{ and } m_p \text{ is virtual} \\ \tau_p + d_p + g & m_k \text{ is } m_j \text{ and } m_p \text{ is virtual} \\ t_k & m_k \text{ has no immediate precedent} \end{cases}$$

These formulae for τ_k are used for the recursive calculation of worst-case D_i and D_j as illustrated in the example shown in Figure 3. For D_i, the worst-case (least D_i) occurs when the clock on the destination node of a virtual message is g time units behind the clock on the source, vice-versa for real messages, and real messages take time less than their d_k to reach the destination node. The exact opposite situation results in the worst-case (i.e., the largest) D_j. Our experience shows that in real systems, it is unusual to find more than one virtual message in a chain of messages, so the effect of virtual messages is usually not as pronounced as shown in this example.

This new causal restriction is enough to ensure proper ordering since no two clocks can be out of sync by more than g. Also, note that since the clocks on the transmitting and receiving nodes can be as much as g apart, m_i may be delivered on the receiving node g time units before its deadline (according to the transmitting node's clock) is reached. This is why D_j must now be $\leq D_i - g$ (instead of $D_j \leq D_i$ as was the case for ideal clocks).

Several questions arise regarding the causal restriction for non-ideal clocks:

1. How much of a negative impact does reducing message deadlines have on network schedulability?

2. How difficult is it to adjust deadlines to conform to this causal restriction?

The first question is investigated further in Section 4 through simulations. Regarding the second question, the difficult task is determining which m_j's precede a given m_i, the subject of the next subsection.

3.3. Adjusting Message Deadlines

Before we can adjust deadlines, we must first identify those m_j's which precede a given m_i. The first step is to figure out all messages to be exchanged between nodes of the system. These can usually be known at design-time, unless the application is as complex and dynamic as an ATM voice/video/data network where new network connections (with dynamic quality-of-service requirements) are set up at run-time. We are not concerned with such complex applications. We focus only on smaller embedded systems where typically all messages (in the worst-case) a node can ever send are known at design-time.

[3] Message m_p is an *immediate* precedent of m_k if $m_p \rightarrow m_k$ and the destination of m_p is the source of m_k.

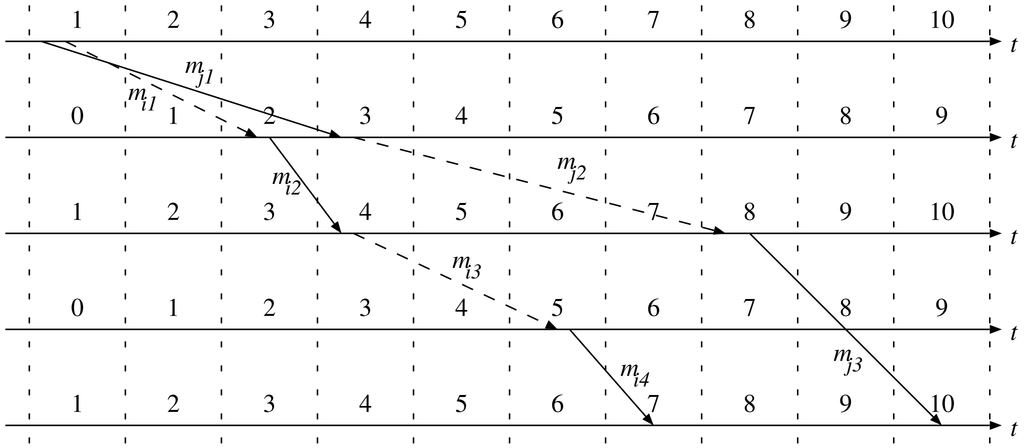

Figure 3. Calculation of D_i and D_j (worst-case). Messages have deadlines $d_{i1} = d_{i2} = d_{i3} = d_{i4} = d_{j1} = d_{j3} = 2$ clock ticks and $d_{j2} = 4$ clock ticks, and the first messages in both chains are sent at $t_0 = 1$. With all real messages, both D_{i4} and D_{j3} would be $t_0 + 8 = 9$, but with virtual messages, D_{j3} is 10 while D_{i4} is only 7. So it is necessary to adjust deadlines to satisfy the causal restriction.

The next step is for the designer to identify the cause-and-effect relations between messages. A CAD tool can greatly help in this. The tool will need to know the following two pieces of information:

- The source and destination processes of each message as well as the message characteristics such as its period and whether it is periodic or sporadic.
- The actuator and sensor pairs such that if the actuator changes the environment, the change is detectable by the sensor. An example would be a heating element and a temperature sensor.

The tool can use this information to suggest to the application designer all possible causal relations between messages. This is best described through an example. Suppose process P_1 uses a periodic message m_1 to control an actuator a. Actions of this actuator can be detected by a sensor s being monitored by process P_2. For example, a may be an industrial drive and s may be a speed sensor. Suppose the system is in a stable state when P_1 announces a mode change. P_1 uses sporadic message m_2 to inform P_2 of the mode change. In the new mode, P_1 uses a different strategy to control a (e.g., the drive may be slowed down) and this change is detected by P_2 through s before it receives m_2 (Figure 4). Since P_2 is still in the old operating mode, it would not expect the drive to slow down and may take some counter-productive actions upon detecting the slow-down.

Notice that m_1 is periodic and m_2 is sporadic and the two have different destinations. This makes it difficult to see that the causal restriction applies to this case because of a virtual message m_v sent from the drive to P_2 through the environment. Following are the steps through which the tool can automatically detect this causal relationship:

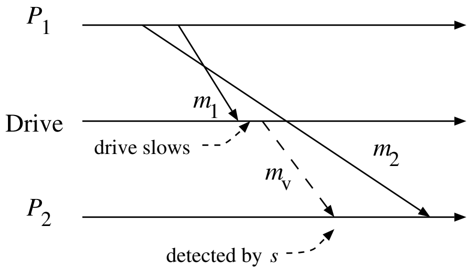

Figure 4. An example where the causal restriction applies because of communication through the environment (broken line).

1. First, the user can specify m_1 which the tool will graphically display for the user (Figure 5). Let $d_1 = 5$ clock ticks.
2. The user will then specify the connection between the actuator a and the sensor s (Figure 6). The time for a "virtual" message to reach its destination will be the time an actuator takes to influence the sensor. For our example, let $d_v = 4$ clock ticks.
3. Since m_1 is periodic, it may be that $m_1 \rightarrow m_v$. The tool will ask the user to confirm or reject this. Confirmation will result in Figure 7.

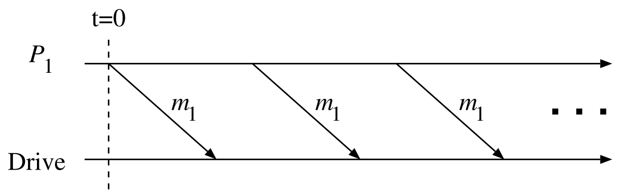

Figure 5. Step 1: specify m_1.

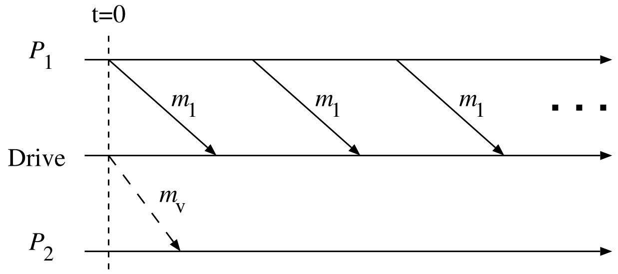

Figure 6. Step 2: specify m_v.

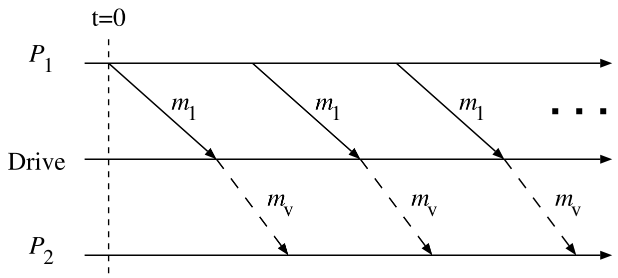

Figure 7. Step 3: confirm causal relation between m_1 and m_v.

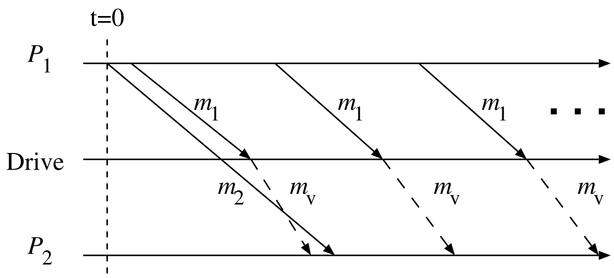

Figure 8. Step 4: specify m_2.

4. Next, the user will specify m_2 and the minimum separation between events `send(`m_1`)` and `send(`m_2`)` (Figure 8). Let this minimum separation be 1 clock tick and $d_2 = 11$ clock ticks.

5. At this stage, the tool can detect that $m_2 \rightarrow m_v$ and `dest(`m_2`)` $\equiv$ `dest(`m_v`)`.

6. Now, the tool must check if the causal restriction is satisfied or not. $D_2 = \tau_2 + d_2 = t_2 + d_2 = 0 + 11 = 11$ clock ticks and $D_v = \tau_v + d_v = \tau_1 + d_1 + d_v = t_1 + d_1 + d_v = 1 + 5 + 4 = 10$ clock ticks. Then the tool will notice that $D_2 \not\leq D_v$ and will alert the user that deadlines must be adjusted.

Remarks:

- In the example, the user specified m_1, m_v, and m_2 in that order. Changing the order will not affect the end result.

- We assumed that the computation delay between receiving a (real or virtual) message and sending another one was 0 (i.e., infinitely fast processors). In practice, the user must specify both minimum and maximum values. The minimum values will be used for D_i and the maximum ones for D_j when $m_j \rightarrow m_i$.

- The graphic display is just to help the user visualize the relation between messages — it is not needed in any of the calculations.

At this point the reader may wonder: if all causal relationships between messages are known, why not program-in this information and use it to order messages (instead of using timestamps) ? The answer is that a message m_j may causally precede m_i at one time and not at another. In the above example, suppose m_2 is sent soon after one invocation of m_1. Then m_2 does not causally precede this invocation of m_1.

Such a tool can use information about individual messages to guide the application designer in recognizing causalities and adjusting deadlines to satisfy the causal restriction. We are currently developing this tool. Still under

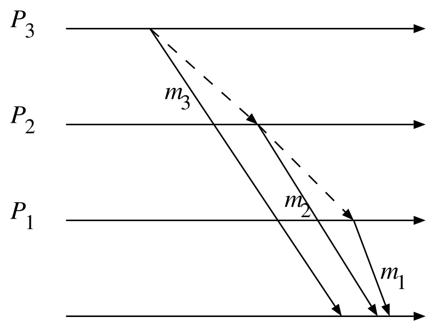

Figure 9. Simulation workload (for $n = 3$).

consideration are several issues such as proving whether or not the causal restriction for non-ideal clocks (Section 3.2) is a necessary condition, and if not, what changes will make it a necessary (optimal) condition.

4. Performance Evaluation

One concern regarding our ordering scheme is that reducing message deadlines according to the causal restriction for non-ideal clocks will reduce network schedulability. We can accept the causal restriction for ideal clocks (and the resulting decrease in message deadlines) as inevitable for correct operation of the system. But the causal restriction for non-ideal clocks reduces deadlines even further. For example, if $m_3 \to m_2 \to m_1$ and these messages are related through virtual messages as shown in Figure 9 and all three have the same destination, then D_2 must be $\leq D_1 - g$ and D_3 must be $\leq D_2 - g$. Using the formulae for τ_k to calculate the D_k, we see that d_2 must be reduced by $2g$. This reduction then affects d_3 which must now be reduced by $4g$. So if there is a chain of n causally-related messages, then the deadline of the first one in the chain may have to be reduced by as much as $2(n-1)g$ (note that n is always finite, i.e., the causal chain is acyclic because it is impossible for a process to send any new information to itself). In this section we use simulations to show that, in fact, there is little or no change in network schedulability when deadlines are adjusted.

Figure 9 shows an example of how the simulation workload is constructed, for $n = 3$. Suppose d_m is the least of the minimum message deadline and the minimum δ in the system. We assign d_m as the deadline of the last message in the chain, m_1. Since the minimum separation between `send(`m_2`)` and `send(`m_1`)` is at least d_m, we have $d_2 \geq d_1 + d_m = 2d_m$ for ideal clocks. Let $d_2 = 2d_m$. This will cause the biggest decrease in d_2 when we make it conform to the causal restriction for non-ideal clocks. So in general, $d_i = id_m$ in accordance with the causal restriction for ideal clocks. Our workload consists of 13 messages. Of these, 12 messages are in 12/n chains, each chain starting at $t = 0$. The 13th message has a period of $1.5d_m$ and a deadline of $1.2d_m$ (without this message, the workload is too degenerate to give any real insight into the problem). We want to see for which minimum d_m is this workload feasibly schedulable when only the causal restriction for ideal clocks is satisfied, and how does this minimum d_m change when deadlines of the 12 causally-related messages are reduced to $d_i = id_m - 2(i-1)g$, $i = 2, 3, \ldots n$ (note that d_1 does not have to be reduced) to conform with the causal restriction for non-ideal clocks.

For simulation we assume a 1 Mbit/s CAN [12] network which is commonly used in embedded systems like automotive and industrial applications. Assume each of the 12 causally-related messages is 79 bits long (32 data bits plus 47 framing bits) and the 13th message is 87 bits long (40 data bits). For CAN networks, clocks can be synchronized to give a g as small as 20μs [20]. In our simulations, we allow for some extra variance and let $g = 30\mu$s. We simulate both the deadline-monotonic (DM) scheduling algorithm [17] and the mixed-traffic scheduler (MTS) algorithm [16] while varying both n as well as d_m. DM is a fixed-priority scheduler which prioritizes messages according to the tightness of their relative deadlines. MTS is a combination of DM and the earliest-deadline (ED) scheduler. ED is a dynamic-priority scheduler which, at any given time, assigns the highest priority to the message with the earliest absolute deadline. MTS combines both ED and DM to give better performance than DM while incurring less overhead than ED.

In our simulations, we first determine the minimum d_m under which a workload is feasibly schedulable when the d_i's satisfy only the causal restriction for ideal clocks. Then, we want to see by how much this minimum d_m has to be increased to make the workload schedulable again when deadlines are decreased to satisfy the causal restriction for non-ideal clocks. If d_m does not change by much, it would mean that schedulability is not affected much when deadlines are decreased, indicating that the causal restriction for non-ideal clocks does not significantly degrade system performance.

Figures 10 and 11 show the simulation results for DM and MTS, respectively, in which for our workloads, the minimum d_m does not change at all under MTS when going from ideal clocks to non-ideal clocks, and increases only minimally under DM. The fact that the minimum d_m changes only slightly (if at all) can be explained as follows. First of all, the causal restriction for non-ideal clocks does not affect the tightest deadline d_1 in the chain. If it did, we would expect a more pronounced negative effect on network schedulability. The second tightest deadline is d_2, which is d_m greater than d_1 and then decreased by $2g$. How much of a negative impact this decrease has depends on how $2g$ compares to d_m (which is the least of the minimum message deadline and the minimum δ). In embedded real-time

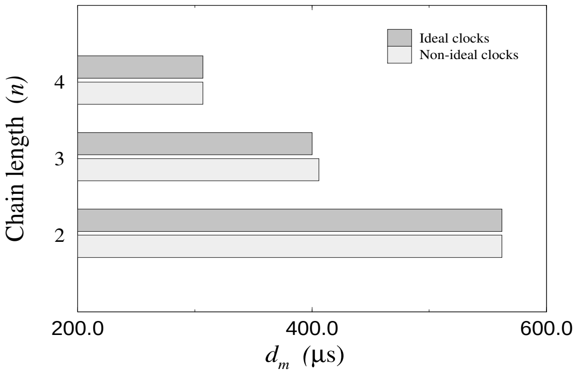

Figure 10. Values of d_m for which workload is infeasible under DM.

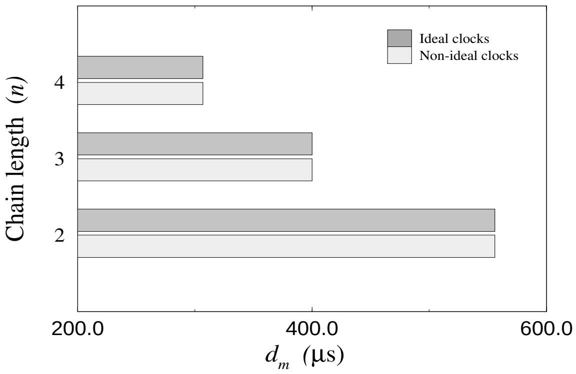

Figure 11. Values of d_m for which workload is infeasible under MTS.

systems, message deadlines are typically at least a few hundred microseconds [16]. The minimum δ must be greater than the response time of the fastest actuator in the system. From Table 1, we see that even the fastest actuators have response times of at least several hundred microseconds. On the other hand, g is usually as small as tens of microseconds [19, 20] for LANs commonly used in embedded systems. So, if $d_1 = d_m = 400\mu$s, then d_2 will be 800μs for ideal clocks which will reduce to 740μs for non-ideal clocks for $g = 30\mu$s, a decrease of only 7.5%. These results show that the causal restriction for non-ideal clocks does not, in any significant way, reduce network schedulability.

Actuator	*Response Time*
Automotive actuators	
Switching time for solenoid valves in ABS [21]	4–10ms
Switching time for fuel injectors [21]	0.65–1.5ms
Motors	
Minimum stepper motor pulse response time [22]	0.83ms (1200 pulses/sec)
Industrial valves	
Solenoid valve (50% step change) [23]	0.6–3s

Table 1. Response times for some commonly-used actuators.

5. Conclusion

Ordering messages according to their causal relationships in a distributed real-time system is necessary to ensure correct and reliable behavior and is also useful for building atomic multicast, replication, and other such higher level services. Causal order must be preserved while ensuring that all messages meet their deadlines — a requirement of all real-time systems. In mass-produced embedded systems such as those used in automotive and industrial control applications, there is another requirement: causal order must be ensured at low overhead to keep costs down as much as possible. Known clock-driven protocols for causal ordering in real-time systems delay all messages for a period Δ greater than the maximum transmission time of any message in the system. If message deadlines are to be met as well, Δ must also be made less than the tightest message deadline. These two conditions on Δ make these protocols too inefficient for low-cost embedded applications. Fortunately, the small scale of such applications presents opportunities for optimization. In this paper, we presented an extension to the Δ-protocol which — instead of delaying all messages for a fixed period Δ — delays each message only until its deadline before delivering it to the application. Our scheme requires that if $m_j \rightarrow m_i$ and the two messages have the same destination, then d_j must be adjusted to satisfy $D_j \leq D_i - g$. We showed that a CAD tool can greatly help the application designer in adjusting deadlines to conform to this requirement. Moreover, we showed that this decrease in deadlines barely affects network schedulability. In this way, we transfer the causal ordering overhead from run-time to design-time and provide a tool to make this task easy for the application designer, thereby reducing the per-unit cost of the system which makes our scheme feasible and attractive for use in small- to medium-sized embedded systems.

References

[1] F. Cristian, H. Aghili, R. Strong, and D. Dolev, "Atomic broadcast: From simple message diffusion to byzantine agreement," in *Proc. Int'l Symposium on Fault-Tolerant Computing*, pp. 200–206, June 1985.

[2] H. Kopetz, "Sparse time versus dense time in distributed real-time systems," in *Proc. Int'l Conf. on Distributed Computing Systems*, pp. 460–467, June 1992.

[3] V. Hadzilacos and S. Toueg, "Fault-tolerant broadcasts and related problems," in *Distributed Systems*, S. Mullender, editor, pp. 97–145, Addison Wesley, New York, second edition, 1993.

[4] L. Lamport, "Time, clocks, and the ordering of events in a distributed system," *Communications of the ACM*, vol. 21, no. 7, pp. 558–565, July 1978.

[5] K. P. Birman, "The process group approach to reliable distributed computing," *Communications of the ACM*, vol. 36, no. 12, pp. 37–53, December 1993.

[6] P. Verissimo, "Ordering and timeliness requirements of dependable real-time programs," *Journal of Real-Time Systems*, vol. 7, no. 2, pp. 105–128, September 1994.

[7] H. Kopetz and K. H. Kim, "Temporal uncertainties in interactions among real-time objects," in *Ninth Symposium on Reliable Distributed Systems*, pp. 165–174, October 1990.

[8] H. Kopetz and K. Kim, "Consistency constraints in distributed real time systems," in *Distributed Computer Control Systems*, M. G. Rodd and T. L. d'Epinay, editors, pp. 29–34, Pergamon Press, 1988.

[9] P. Verissimo, "Real-time communication," in *Distributed Systems*, S. Mullender, editor, pp. 447–490, Addison Wesley, New York, second edition, 1993.

[10] D. R. Cheriton and D. Skeen, "Understanding the limitations of causally and totally ordered communication," in *Proc. ACM SIGOPS*, pp. 44–57, December 1993.

[11] P. Verissimo, "Causal delivery protocols in real-time systems: a generic model," *Journal of Real-Time Systems*, vol. 10, no. 1, pp. 45–73, September 1996.

[12] *Road vehicles — Interchange of digital information — Controller area network (CAN) for high-speed communication. ISO 11898*, 1st edition, 1993.

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

[14] S. Mullender, editor, *Distributed Systems*, Addison Wesley, New York, second edition, 1993.

[15] P. Verissimo, P. Barnett, P. Bond, A. Hilborne, L. Rodrigues, and D. Seaton, "Extra Performance Architecture (XPA)," in *Delta-4: A Generic Architecture for Dependable Distributed Computing*, D. Powell, editor, pp. 211–266, Springer Verlag, 1991.

[16] K. M. Zuberi and K. G. Shin, "Non-preemptive scheduling of messages on Controller Area Network for real-time control applications," in *Proc. Real-Time Technology and Applications Symposium*, pp. 240–249, May 1995.

[17] K. Tindell, A. Burns, and A. J. Wellings, "Calculating Controller Area Network (CAN) message response times," *Control Engineering Practice*, vol. 3, no. 8, pp. 1163–1169, 1995.

[18] C.-C. Han and K. G. Shin, "Real-time communication in FieldBus multiaccess networks," in *Proc. Real-Time Technology and Applications Symposium*, pp. 86–95, May 1995.

[19] H. Kopetz and W. Ochsenreiter, "Clock synchronization in distributed real-time systems," *IEEE Trans. on Computers*, vol. 36, no. 8, pp. 933–940, August 1987.

[20] M. Gergeleit and H. Streich, "Implementing a distributed high-resolution real-time clock using the CAN-bus," in *Proc. 1st International CAN Conference*, September 1994.

[21] K. Muller, "Actuators," in *Automotive Electronics Handbook*, R. K. Jurgen, editor, pp. 10.1–10.34, McGraw-Hill, New York, NY, 1995.

[22] M. S. Sarma, *Electric Machines: Steady-State Theory and Dynamic Performance*, West Publishing Company, St. Paul, MN, second edition, 1994.

[23] V. Liantonio, "Conceptual advancements in the solenoid powered control valve art," in *Developments in Valves and Actuators for Fluid Control*, D. R. Airey, editor, pp. 25–36, Scientific And Technical Information, Oxford, 1990.