Monitoring Timing Constraints in Distributed Real-time Systems

Sitaram C. V. Raju *

Department of Computer Science and Engineering
University of Washington
Seattle WA 98195

Ragunathan Rajkumar†
Farnam Jahanian

IBM T. J. Watson Research Center
P.O. Box 704
Yorktown Heights, NY 10598

Abstract

In this paper, we describe a run-time environment for monitoring distributed real-time systems. In particular, we focus on the problem of detecting violations of timing assertions in an environment in which the real-time tasks run on multiple processors, and timing constraints can be either inter-processor or intra-processor constraints. Constraint violations are detected at the earliest possible time by deriving and checking intermediate constraints from the user-specified constraints. If the violations must be detected as early as possible, then the problem of minimizing the number of messages to be exchanged between the processors becomes intractable. We characterize a subclass of timing constraints that occur commonly in distributed real-time systems and whose message requirements can be minimized. We also take into account the drift among the various processor clocks when detecting a violation of a timing assertion. Finally, we describe an implementation of a distributed run-time monitor.

1 Introduction

As embedded real-time systems become more sophisticated, the ability of the system to provide dependable and timely service becomes critical. The unpredictability of the physical environment and the inability to satisfy design assumptions can cause unexpected conditions or violations of system constraints at run-time. It is highly desirable that under these error conditions, total system failure does not occur and critical system functions of the system are still performed. In other words, constraint violations must be detected and appropriate action taken. For example, an estimate on the worst-case execution time of a task or a minimum interarrival time for an asynchronous signal may be violated at run-time. Despite the contributions of formal verification methods and recent real-time scheduling results, the need to perform run-time monitoring of these systems is not diminished due to the inherent complexity of these systems and the non-determinism in dealing with the external environment. In earlier work [2], we have presented a general framework for formal specification and monitoring of run-time constraints in time-critical systems. We also described a single-processor implementation of a monitoring subsystem for an IBM RS/6000 workstation running the AIXv.3 operating system[1]. In this paper, we consider the problem of run-time monitoring in a distributed real-time system.

Monitoring a timing constraint becomes more complicated in a distributed system due to the occurrences of events on multiple processors. This has several implications. First, detecting a violation *as early as possible* may require partial evaluation of a timing constraint as time progresses. Secondly, extra messages may have to be exchanged to propagate the occurrence of an event on a processor to others. Finally, in the absence of perfectly synchronized processor clocks or a global system clock, the meaning of a timing assertion on distributed events must be defined precisely.

Our run-time monitor for distributed real-time systems is based on the Real-time Logic (RTL) model proposed in [7], and our prototype is an extension of the uniprocessor implementation of [2]. In this model, a system computation is viewed as a sequence of event occurrences. The design assumptions and system properties that must be maintained are expressed as invariant relationships between various events, which are monitored during run-time. If a violation of an invariant is detected, the system is notified so that suitable recovery options can be taken. The invariants are specified using a notation based on RTL, and timing constraints are allowed to span processors. Our run-time monitoring facility monitors and detects violations in a distributed fashion. This distribution prevents any single monitoring process from becoming a bottleneck. In addition, monitoring of events on the processors where they occur allows violations to be detected as early as possible. A clock synchronization algorithm ensures that event occurrence times on different processors can be meaningfully compared.

1.1 Related Work

Despite extensive work on monitoring and debugging facilities for parallel and distributed systems, run-

*Supported in part by the Office of Naval Research under grant number N00014-89-J-1040 and by NSF under grant number CCR-9200858.

†Currently at Software Engineering Institute, Carnegie-Mellon University.

[1]RS/6000 and AIX are trademarks of IBM Corporation.

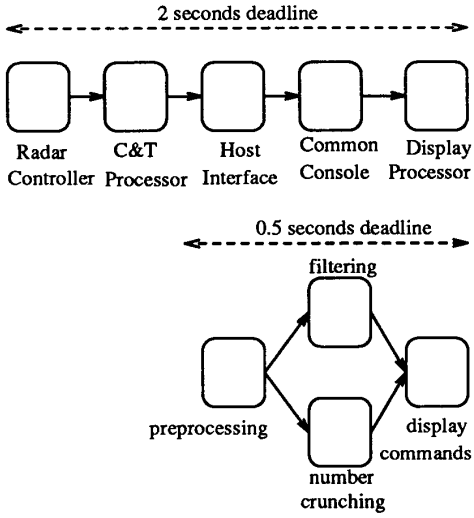

Figure 1: An Aircraft Tracking System

time monitoring of real-time systems has received little attention with a few exceptions. Special hardware support for collecting run-time data in real-time applications has been considered in a number of recent papers [6, 15]. These approaches introduce specialized co-processors for the collection and analysis of run-time information. A related work [5] studies the use of monitoring information to aid in scheduling tasks. A work closer to our approach is a system for collection and analysis of distributed/parallel (real-time) programs [9]. The work is based on an earlier system for exploring the use of an extended E-R model for specification and access to monitoring information at run-time [12]. A real-time monitor developed for the ARTS distributed operating system is presented in [14].

The rest of this paper is organized as follows. Section 2 describes an aircraft tracking system and discusses issues in detecting a violation of a timing assertion. Section 3 presents our event-based computation model, and discusses specification of timing assertions. Section 4 presents a solution to the problem of minimizing extra messages to propagate event occurrences to other processors, and discusses the effect of synchronized clocks on detecting a violation. Section 5 describes a prototype implementation of a monitor for a network of RS/6000s running AIXv.3 operating system. Finally, section 6 presents our concluding remarks.

2 Motivation and Research Issues

In a distributed real-time system, there can be timing constraints imposed on events across multiple processors. These interprocessor timing constraints are among the hardest to enforce as well as to verify. This section motivates the problem by describing an aircraft tracking system with end-to-end requirements.

2.1 Aircraft Tracking System

The structure of an aircraft tracking system is shown in Figure 1. A radar signal is received by a radar controller, which is a special-purpose processor. The signal is fed into a general-purpose processor that does calibration and tracking. Next, the signal is sent to a host interface that does some preprocessing and sends the signal to a console processor[2], which performs some filtering and number-crunching in parallel. Finally, the signal is sent to a special-purpose display processor that displays the appropriate tracking information on the monitor. There is a 2 seconds end-to-end deadline from the time an event is received by the radar controller to the time it is displayed by the display processor. In addition, there is an intermediate 0.5 second deadline on the *display commands* step on the display processor from the *preprocessing* step on the host interface processor.

2.2 Issues in Monitoring of Distributed Constraints

Several issues need to be addressed by a distributed monitor that checks interprocessor timing constraints.

- *Time of detection of violations:* Detecting violations as early as possible is a desirable property because it can allow the system to take corrective action before the violation actually happens. We address this issue in Section 3.4.
- *Number of messages:* Since events happen on different processors and timing constraints can span processors, some form of interprocessor communication is needed to propagate this information. In a distributed environment, event occurrences must be communicated using messages. Minimizing the number of extra messages is crucial for reducing overhead. We address this issue in Section 4.1.
- *Clocks and Timer Granularity:* When an event occurs there must be a way of recording the occurrence time of the event. The granularity of timestamping determines the minimum observable spacing between 2 consecutive events on a processor. Timestamping is typically done by reading the clock on the local processor. A distributed system must also deal with the fact that the clocks on different processors are not perfectly synchronized. The processor clocks, however, can be kept synchronized within a known maximum bound on the deviation between them. Clock synchronization allows controlled comparison of timestamps from different clocks. In particular, one must take into consideration the deviation between clocks when evaluating a timing assertion at run-time. We address this issue in detail in Section 4.2.
- *Resource management:* Another fundamental aspect of a distributed monitor involves the need to

[2]In a real system, the signal may be sent to one of many consoles. In this paper, we assume a single console for the sake of simplicity.

quantify the timing intrusiveness of the monitoring activities on the timing behavior of the real-time application. A discussion of our approach to scheduling the monitoring activities is beyond the scope of this paper and interested readers are referred to [8]. This approach allows the run-time monitoring processes to be scheduled as time-constrained activities and therefore can be part of the schedulability test for the system. In addition, it may be necessary to record multiple occurrences of an event to detect timing violations. The size of an event history, however, must be bounded to guarantee a worst-case time bound on insertions/deletions. We assume that an approach that bounds the number of occurrences of an event is being used as advocated in [7].

2.3 Approach

Run-time monitoring of a system requires timestamping and recording of the relevant event occurrences, analyzing the past history as other events are recorded and providing feedback to the rest of the system. In different situations, one can envision a monitoring system that provides feedback to the operator, the application tasks or the scheduler at run-time. Our goal is to monitor constraints that may span several processors and to detect and notify a violation as early as possible.

Our distributed monitor consists of a set of cooperating monitor processes (daemons), one on each processor. Application tasks on a processor inform the local monitor daemon of events as they occur. The monitor daemon on a processor checks for a violation as local events happen; it also sends the information about certain event occurrences that are needed by remote monitors to other processors. The monitor daemons detect a violation of a timing constraint in distributed fashion.

3 Timing Constraints in Real-Time Systems

We express the timing constraints in a notation based on RTL [7, 2]. In this section, we present an informal overview of the computational model and then discuss the representation of a timing assertion as a directed constraint graph. Finally, we introduce the conditions under which a timing assertion can be violated.

3.1 Specification of Timing Assertions

An application consists of a set of cooperating tasks, perhaps running on multiple processors. The monitoring subsystem views a computation of the system as a partially-ordered sequence of event occurrences. Informally, events represent things that happen in a system. For example, an event may denote the start/completion of a program segment, reading a new sensor value into a program variable or receiving a message from another task. An event occurrence defines a point in time at which a particular instance of the event happens in a computation. Thus, a *safety* property or a timing constraint can be expressed as an assertion about the relationship between event occurrences in a computation. For example, suppose event E denotes the invocation of a task T and another event E' denotes the completion of the task, and task T is executed periodically. Each invocation of the task corresponds to an occurrence of the event E. Similarly, when a particular invocation of task T completes, it corresponds to an occurrence of event E'. A timing constraint, such as a deadline, specifies the relationship between the occurrences of event E and E'. A deadline of $100ms$ on the execution of task T requires that the corresponding occurrences of E and E' should be within 100ms.

To detect timing constraints, it may be necessary to record multiple occurrences of an event. For example, to check a constraint on the interval between two successive instances of an event, the past two instances of the event must be stored. The occurrences of an event are stored in a circular queue called its *event history*, which maintains the last n occurrence times of the event. The value of n depends on the assertion.

We provide two functions for accessing the event histories: the occurrence function $@(e,i)$, which returns the time of the ith occurrence of event e, and $@val(v,i)$, which returns the value of the ith assignment to a variable v. A positive occurrence index is absolute with respect to the beginning of the computation sequence. For example, $@(e,5)$ refers to the 5th occurrence of event e. When the index i is negative, it refers to the ith most recent occurrence of the event in a computation. For example, $@(e,-1)$ denotes the time of the most recent occurrence of e. An occurrence index of 0 is undefined. We present two examples to illustrate how the notation can be used.

Example: Consider two events e_1 and e_2 which must always occur in pairs and within 5 time-units of each other. The following formula specifies such a constraint.

$$\forall i \;\; @(e_1,i) \leq @(e_2,i)+5 \vee @(e_2,i) \leq @(e_1,i)+5 \quad (1)$$

Example: The value of the occurrence indices in a timing constraint need not be a variable or a positive integer. It can be relative to the current index in a computation. Consider an event whose successive occurrences must be separated by at least 5 time-units. The following RTL formula specifies such a constraint.

$$@(response,-2) \leq @(response,-1) - 5$$

If this constraint is checked whenever a *response* event occurs, then it is equivalent to

$$\forall i > 1 \; @(response,i-1) \leq @(response,i) - 5$$

3.2 Graph Representation of Timing Constraints

If a timing assertion is in a disjunctive normal form as in Equation 1, each conjunct can be represented as a directed, weighted graph, called a *constraint graph*. Each constraint graph represents a conjunction of predicates, and each edge in the graph is a predicate of the form:

$$@(e,i) \leq @(f,j) \pm C$$

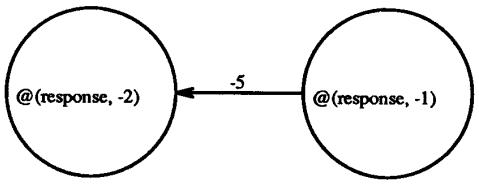

Figure 2: Delay constraint

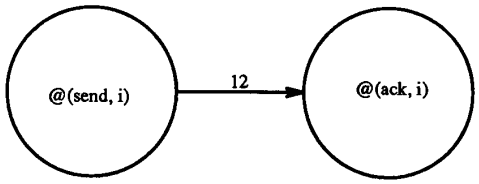

Figure 3: Deadline constraint

such that i, j are integer variables/constants[3] and C is an integer constant.

Intuitively, a predicate in a conjunct represents either a delay or a deadline constraint on a pair of events. The vertices of the constraint graph correspond to unique occurrence functions; the weighted edges denote the constraints between event pairs. For example, the following *delay constraint*:

$$@(response, -2) \leq @(response, -1) - 5$$

is represented by an edge with weight *-5* as shown in Figure 2. A predicate of the form $@(response, 1) \leq C$ where C is an absolute time value is translated to an edge $\mathbf{0} \xrightarrow{C} @(response, 1)$ where **0** is a special "zero vertex" designed to take care of constants. Similarly, a predicate of the form $C \leq @(response, 1)$ is represented by an edge $@(response, 1) \xrightarrow{-C} \mathbf{0}$.

As an example of a *deadline constraint*, consider the following assertion:

$$\forall i \ @(ack, i) \leq @(send, i) + 12$$

This constraint specifies that an *ack* event must occur within 12 time-units of its corresponding *send* event, and is represented by an edge with positive weight *12* as shown in Figure 3.

A *path* between two vertices u and v in the graph is a sequence of edges from u to v. The *length* of a path is the sum of the weights of all edges along the path. In the rest of the paper, without loss of generality, we do not associate any occurrence indices to events. This is possible because a specific constraint graph is instantiated for a set of event occurrence indices before checking for a violation.

3.3 Implicit Constraints

In addition to the explicit delay or deadline edges in a constraint graph, we can derive certain implicit constraints often as an intermediate deadline or delay.

[3] We assume that indices are not arithmetical expressions.

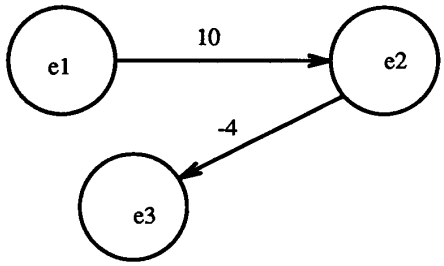

Figure 4: Intermediate deadline from *e1* to *e3*

In fact, it is possible that an implicit constraint is violated before an explicit deadline or delay becomes unsatisfiable at run-time.

For example, consider the simple constraint graph in figure 4. It consists of two explicit timing constraints: a deadline edge and a delay edge. Events *e1*, *e2* and *e3* occur on processors 1, 2 and 3, respectively. There is an explicit deadline from *e1* to *e2*. In addition, since there is a path from *e1* to *e3* of length 6, there is an implicit, intermediate deadline of 6 from *e1* to *e3*. If the intermediate deadline is not met, then either the explicit deadline or the delay constraint from *e3* to *e2* will eventually be violated. If the violation of the implicit constraint between *e1* to *e3* is detected, the system can be notified before any of the two user-specified constraints are violated. As a result, corrective action can potentially be taken even before the application-level constraint is violated.

3.4 Checking Constraint Graphs

The constraint graphs must be checked for potential violations at certain discrete points in time. We establish these discrete points of time in this section.

When an event occurs that may affect the satisfiability of a timing assertion, a satisfiability checker is invoked to check for violations. The checker instantiates the vertices of the graph from event histories. For example, the vertex *@(response, -1)* will be replaced by the occurrence time of the most recent *response* event. The vertex *@(send, i)* will be replaced by the occurrence time of the current activation of *send* event. Vertices that have been instantiated are merged with the **0** vertex as follows: Every edge with weight w incoming to a vertex, to which a time value t has been assigned, is replaced with an edge with weight w-t incident on the **0** vertex. Conversely, every edge with weight w outgoing from a vertex, to which time t has been assigned, is replaced with an edge of weight w+t outgoing from the **0** vertex. Vertices that have not yet occurred are not instantiated. Instead, an edge from the uninstantiated vertex to the **0** vertex, of weight equal to *-current_time* is added. This is an assertion that the event has not happened since system startup. The actual algorithm for checking violations will be discussed in section 4.2.

We state a lemma that establishes the conditions under which a timing assertion (constraint graph) may be violated.

Lemma 1 In a constraint graph, the earliest time a constraint can be violated is as follows:

1. A delay constraint will be violated, if for a path of negative length $-T$ ($T \geq 0$) from vertex e_n to vertex **0**, the event corresponding to vertex e_n happens before time T.

2. A deadline constraint will be violated if the minimum length T ($T \geq 0$) of all shortest paths from vertex **0** to all other vertices is to a vertex e_m and the event corresponding to vertex e_m does not happen at or before T.

Proof: The proof has been omitted due to space limitations; a detailed proof can be found in [11].

Lemma 1 states that delay violations need only be tested whenever an event occurs, and deadline violations need not be tested before some timeout value. Hence, a constraint graph must be checked for violations after the occurrence of any event in the graph. The event occurrence is instantiated in the graph with its occurrence time and the graph is checked for violations. If the graph is not violated, the length of the minimum of the shortest paths from vertex **0** to all uninstantiated vertices is computed. If this length P is not infinity then a timer that expires at time P is set. The graph is again checked for violations when the timer expires or when an event happens, whichever is earlier.

4 A Monitor for Distributed Real-Time Systems

In this section, we focus on our approach to deal with the issues that arise in monitoring distributed real-time systems such as the aircraft tracking system of Figure 1.

We assume that interprocessor communication is reliable. This assumption is valid whenever a reliable communication mechanism based on acknowledgments is used. We also assume that there is no migration of application tasks among processors. This assumption can be relaxed if as part of a mode change, the constraints to be monitored and the new communication patterns are also re-established.

4.1 Minimization of Messages

Messages must be passed across processor boundaries to check interprocessor timing constraints. In this subsection, we address the issue of minimizing these message-passing requirements. We assume that each monitor process knows every constraint graph that contains one or more events happening on its local processor. Whenever a local event occurs, a monitor process must decide whether the event must be communicated to other monitors. We also assume in this subsection that there is a single clock in the system. This assumption will be relaxed in the next subsection.

We shall use the following terminology. There is a correspondence between an event e_n, the processor n on which the event happens and the monitor on processor n. Hence, we will use the phrase e_i*'s monitor* to mean the monitor local to the processor on which e_i occurs.

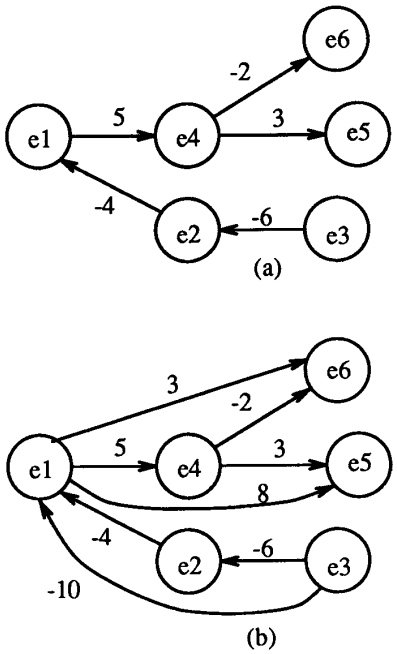

Figure 5: Determining the recipients of event e_1(a) The application-level constraint graph. (b) The shortest path constraint graph for e_1.

Given a constraint graph G and a vertex e_i, the list of monitors to whom the occurrence of e_i must be communicated can be determined as follows. Run the shortest-path algorithm on the graph G such that the shortest path from any vertex to every other vertex is obtained. We refer to this resulting graph as the *shortest path graph*. This transformation of the constraint graph adds edges that represent *implicit constraints*. Messages with event occurrence times may also need to be sent over these additional edges. The shortest path graph captures both explicit and implicit constraints. Since implicit constraints are derived from explicit constraints, the violation of an implicit constraint implies the (potentially future) violation of an explicit constraint. To detect violations as early as possible, implicit constraints need to be considered.

Whenever an event e_i occurs, its occurrence needs to be communicated (directly or indirectly) to the monitor of any vertex e_j, if in the shortest path graph, there exists a path with positive weight from e_i to e_j or a path with negative weight from e_j to e_i. This procedure is illustrated in Figure 5(a), which shows a constraint graph G. The shortest-path graph derived for *one* vertex e_1 is shown in Figure 5(b). From this graph, whenever event e_1 occurs, it must be communicated (directly or indirectly) to the monitors of events e_2 through e_6.

Using the shortest path graph to determine the recipient monitors of an event occurrence can be very pessimistic. It is often possible that some of the messages can be eliminated as they are either redundant or 2 (or more) messages can be combined into a single message. For example in Figure 5(b), e_1's occurrence needs to be communicated to e_4's monitor and to e_5's monitor. Also, e_4's occurrence time needs to be communicated to e_5's monitor. However, the weight of the edge from vertex e_1 to vertex e_5 is the same as the length of the path $e_1e_4e_5$. As a result, the message from e_1's monitor to e_5's monitor, containing the occurrence time of e_1, can be eliminated.

Again in Figure 5(b), e_1's occurrence time needs to be sent to the monitors of e_2 and e_3. In addition, e_2's occurrence time needs to be sent to e_3's monitor. There is also an ordering of events, e_1 (first), e_2 (second) and e_3 (last) such that if these events happen in any other order, a constraint would be violated. As a result, the message from e_2's monitor to e_3's monitor can also carry the occurrence time of e_1. Thus, the message from e_1's monitor to e_3's monitor can be eliminated.

Naturally, it is desirable that the maximum number of messages are either removed or combined. However, the problem of minimizing the number of messages for arbitrary constraint graphs is intractable. We show next that removing the maximum number of redundant messages is NP-complete for constraint graphs whose edges have positive weights only. The formal statement of the problem called irredundant deadline graph (IDG) problem is as follows:

Instance: Given a constraint graph G with positive weights for all edges, and a positive integer $K \leq$ number of edges in G.

Question: Is there a subset $G' \subseteq G$ where the number of edges in G' is $\leq K$ such that, for every ordered pair of vertices, $u, v \in G$, the shortest path from u to v in graph G' is of length d, if and only if the shortest path from u to v in G is also of length d ?

Theorem 1 Irredundant deadline graph (IDG) is NP-complete.

Sketch of proof: By transformation from the Minimum Equivalent Digraph problem (MED) [4]. The transformation is to assign a weight of 0 to every edge of an instance of MED. A detailed proof can be found in [11].

Given that the problem of minimizing the number of messages is intractable, we next consider sub-classes of constraint graphs that are likely to occur in real-time systems, and whose message requirements can be easily minimized.

Definition: An event e_i is said to *precede* event e_j in a constraint graph, if there exists a path from vertex e_j to vertex e_i that consists of delay edges (i.e., edges with negative weights) *only*.

Given this definition, we use the terms "delay edges" and "precedence edges" interchangeably.

4.1.1 Precedence-Preserving Graphs

The delay edges impose precedence or partial ordering on the set of event occurrences in a constraint graph. Intuitively, the delay edges may represent the computation time of a task or causality between a pair of events. For example, if there is a delay from event e_1 to e_2 (a negative edge from e_2 to e_1), event e_1 must occur before (precede) e_2. Otherwise, the delay constraint is violated. In real-time systems, it is common to have an event that triggers a task execution, which in turn generates other events. For example, in the aircraft tracking system described in Section 2, the computation in each module triggers the computation in the next module on completion. An incoming track signal is processed by different modules in pipelined fashion. As a result, every data item that crosses module/processor boundaries has delay or precedence constraints, and deadline constraints tend to be end-to-end deadlines.

In general, precedence relations are natural in real-time systems where data streams are processed in pipeline stages. Consider a distributed audio/video system that processes its data in pipelined stages from one node to another node via gateways and communication networks. More specifically, the audio/video signal is digitized and compressed at a sender node and transmitted to a receiver node where it is uncompressed and displayed. The audio/video data must be received/displayed at the receiver node at a precise rate. Hence, there are precedence constraints between the various stages. If the data corresponds to live interaction such as video conferencing, latency requirements will force an end-to-end deadline as well.

Precedence relations among events also exhibit desirable properties from the communication requirements viewpoint. For example, if event e_1 precedes event e_2, then e_2's monitor must receive the occurrence time of e_1 before e_2 occurs. Otherwise a violation has taken place. Hence, e_2's monitor always has the potential to send the occurrence time of e_1 and e_2 in a single message to a third monitor. These combinations can save messages. We present a sub-class of constraint graphs called precedence-preserving graphs where sending messages only along the delay edges is sufficient to detect all violations at the earliest possible time.

Definition: A *precedence-preserving graph* is a constraint graph that satisfies the following condition: If there is a shortest path from vertex e_i to vertex e_j of positive length, then the source vertices of deadline edges on the path precede e_j.

Recall the aircraft tracking system described in Section 2.1. The corresponding constraint graph, shown in Figure 6, is a precedence-preserving graph. The 2 seconds deadline edge is between the RC and DP/cc vertices, where the former precedes the latter. Also, the 0.5 seconds deadline is between the HI/pp and DP/cc vertices, which also have a precedence ordering. As a result, in this graph, messages need be sent only along the delay edges. For example, the node RC will send its event *only* to node CT. If the shortest path graph were used, there exists a positive path

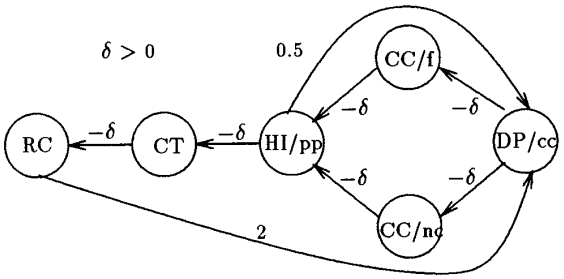

Figure 6: The Constraint Graph for the Tracking System of Figure 1

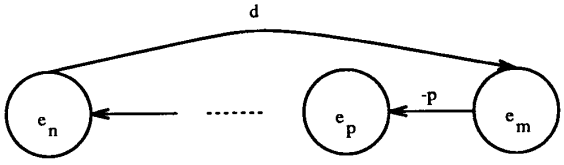

Figure 7: Case where e_n precedes e_m

from *RC* to every other node so that *RC* would have to send 5 messages. The precedence-preserving graph thus results in substantial savings of messages without compromising the time at which a violation will be detected. We now prove this property of precedence-preserving graphs. The theorem is based on the assumption that the message communication time between any two processors is less than the delay constraints between events on the two processors. We also assume that a message carries the time of occurrence of the local event and its predecessor events.

Theorem 2 In a precedence-preserving graph, if messages are being sent only along the precedence edges, then all violations will be detected at the earliest possible time.

Proof: The proof consists of showing that messages along precedence edges carry the required information of event occurrences to a vertex before delay or deadline violations can occur.
Delay violation: Consider a vertex e_n in a precedence-preserving graph. A delay constraint to another vertex e_d can be of 2 types:

1. there is a negative edge from e_d to e_n.
2. there is a path containing negative edges only from e_d to e_n.

In the first case the occurrence time of e_n is sent to e_d. Delay violations can be detected according to Lemma 1. In the second case there are 2 possibilities:

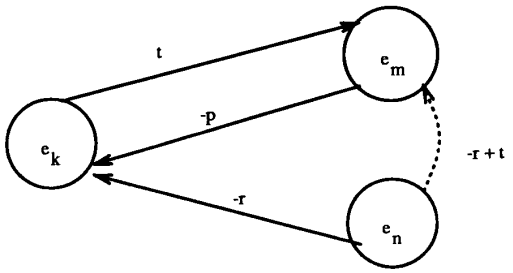

Figure 8: Case where e_k precedes e_n and e_m

1. The immediate predecessor of e_d in the constraint graph has already occurred. Hence e_d will have information about the occurrence time of the immediate predecessor and its predecessors. Hence, constraint violations, if any, can be detected by Lemma 1.
2. The immediate predecessor of e_d, say e_i, has not happened. This implies that the delay constraint to e_i has been violated. Hence the occurrence time of e_i and its predecessors is irrelevant.

Deadline violation:

Lemma 2 If there is a deadline from vertex e_n to vertex e_m, then either e_n precedes e_m or there exists a vertex e_k that precedes both e_n and e_m.

Proof of lemma 2: Consider the path from e_n to e_m. Let the path be e_n, e_i, e_{i+1}, e_{i+2} $\ldots e_{i+p}$, e_m. If the edge from e_n to e_i is positive, then from the definition of precedence-preserving graphs, e_n must precede e_m. If the edge from e_n to e_i is negative, then consider the first positive edge in the path (there must be such an edge as sum of weights on the path is positive). Say the positive edge is from e_{i+a} to e_{i+a+1}. Now, from the definition of precedence oriented graphs, e_{i+a} precedes e_m. Also since there is a sequence of negative edges from e_n to e_{i+a}, e_{i+a} precedes e_n. Thus, e_{i+a} is the vertex e_k.

Q.E.D.

Consider the constraint graph of Figure 7 where e_n precedes e_m . Let e_p be the immediate predecessor of e_m. Let $-p$ $(p > 0)$ be the weight of edge from e_m to e_p. Hence there is a deadline $d - p$ on e_p. Since,

$$d - p < d$$

the deadline on e_p is shorter than the deadline on e_m. Hence the occurrence time of e_n needs to be communicated to e_p first. When e_p occurs, the occurrence time of e_p and its predecessors events will be communicated to e_n. By extending this argument to $e_p's$ immediate predecessor and so on, it can be seen that messages need to be sent along precedence edges only.

Now consider the constraint graph of Figure 8 where a vertex e_k precedes e_n and e_m. Consider the deadline from e_n to e_m.

$$\text{deadline} = @(e_n) - r + t$$

The deadline from e_k to e_m is

$$@(e_k) + t$$

Assume,

$$@(e_n) - r + t < @(e_k) + t$$

then,

$$@(e_n) < @(e_k) + r$$

But this means that the delay constraint from e_k to e_n is violated and it will be detected by e_n. In the absence of violation, the deadline on e_m is not shortened. Hence, messages along the precedence edges will suffice.

Q.E.D.

If a constraint graph is *not* a precedence-preserving graph, its message requirements cannot be (easily) minimized but they can still be reduced. An edge E from vertex e_i to vertex e_j can be deleted from a shortest-path constraint graph G_s, if the weight on the edge is greater than or equal to the shortest path from e_i to e_j in the graph $G_s - E$. If an edge can be removed, then the corresponding message along the edge need not be transmitted. A more general description is given in [11].

4.2 Effect of Approximately Synchronized Clocks

The occurrence time of an event corresponds to the local time at its time of occurrence on the processor where the event occurs. In a distributed system, the clocks on different processors are not identical and deviate from one another. As a result, the checking of constraint graphs must take these clock deviations into account.

If the deviation between various clocks is not bounded by a known value, it is difficult (perhaps impossible) to enforce in a meaningful way a timing constraint whose events span multiple processors. We therefore use a clock synchronization algorithm [1, 3, 10] to bound the deviations among the various processor clocks. A clock process on each processor synchronizes with the clock process on a master processor (master clock) by exchanging messages.

Let ϵ be the maximum deviation among the clocks. We now state a theorem that gives the necessary and sufficient condition for violations in a constraint graph in which a constraint spans events on more than one processor. The intuition behind the theorem is as follows: If there is a constraint of length P between events on two different processors, the constraint length should be changed to $P - \epsilon$ within the system. For example, if event e_2 must happen within 10 time-units after event e_1, the bounded clock deviation requires that e_2 actually happen within 10-ϵ after e_1. Otherwise, a violation may have occurred.

Theorem 3 Given a cycle spanning more than one processor in a constraint graph, there exists a set of clock deviations for which a constraint is violated, if and only if the length of the cycle is less than ϵ.

Proof: Since there is a cycle spanning more than one processor, there must exist two adjacent vertices e_i and e_j in the cycle that are on two different processors.

If part. Let there be a cycle of length $< \epsilon$. We consider delay and deadline constraints between e_i and e_j, and show that in either case the constraint is violated. The occurrence time of any event is the time of its occurrence according to its local clock. Also, the local clock of an event refers to the clock of the processor on which the event happens.

1. Delay constraint of P $(0 < P)$ between e_i and e_j, i.e. e_j must occur any time *at* or *after* P time-units after the occurrence of e_i. This means that

   $$@(e_i) + P \leq @(e_j)$$

   Let the local clock of e_j be ϵ ahead of the local clock of e_i. A violation takes place if e_j occurs on e_j's processor any time before $P + \epsilon$ after e_i's occurrence. Hence, the constraint between the events must be adjusted as

   $$@(e_i) + (P + \epsilon) \leq @(e_j) \quad (2)$$

   Since there is a cycle, there has to be a path from e_i to e_j. Let the length of this path be Q. That is, e_j must occur any time *at* or *before* Q time-units after the occurrence of e_i. This means that

   $$@(e_j) \leq Q + @(e_i) \quad (3)$$

   For Equations (2) and (3) to be satisfied, we must have

   $$Q - P \geq \epsilon \quad (4)$$

   This contradicts our assumption that the cycle length is less than ϵ, i.e. $Q - P < \epsilon$.

2. Deadline constraint of P $(0 < P)$ between e_i and e_j, i.e. e_j must occur any time *at* or *before* P time-units after the occurrence of e_i. This means that

   $$@(e_j) \leq @(e_i) + P \quad (5)$$

   Let the local clock of e_j be ϵ behind the local clock of e_i. A violation takes place if e_j occurs any time after $P - \epsilon$ after e_i. Hence, the constraint between the two events must be adjusted as

   $$@(e_j) \leq @(e_i) + (P - \epsilon) \quad (6)$$

   Since there is a cycle, there has to be a path from e_j to e_i of length $-Q$. This means that,

   $$@(e_i) + Q \leq @(e_j) \quad (7)$$

   For Equations (6) and (7) to be satisfied, we must have

   $$P - Q \geq \epsilon \quad (8)$$

   This contradicts our assumption that the cycle length is less than ϵ, i.e. $P - Q < \epsilon$.

Only if part. Let a constraint between vertices e_i and e_j be violated. The constraint can either be a delay or a deadline constraint. We consider the 2 cases separately and show that in either case there will be a cycle of length $< \epsilon$ in the constraint graph.

1. Delay constraint of P $(0 < P)$ between e_i and e_j. Since this has been violated, the separation between e_i and e_j is less than P, i.e.,

   $$@(e_j) - @(e_i) < P$$

   Since the local clock of e_j can be ahead of the local clock of e_i by a maximum of ϵ, the constraint can be violated if

   $$@(e_j) - @(e_i) < P + \epsilon$$

   or,

   $$@(e_j) < @(e_i) + P + \epsilon$$

   This implies that there is a path from e_i to e_j of length Q, where $Q < P + \epsilon$. Thus there is a cycle of length

   $$= Q - P < P + \epsilon - P < \epsilon$$

2. Deadline constraint of P $(0 < P)$ between e_i and e_j. Since this has been violated, the distance between e_i and e_j is greater than P, i.e.,

   $$@(e_j) > @(e_i) + P$$

   Since the local clock of e_j can be behind the local clock of e_j by a maximum of ϵ, the constraint can be violated if

   $$@(e_j) > @(e_i) + P - \epsilon$$

   This constraint implies that there is a path from e_j to e_i of length $-Q$, where $Q > P - \epsilon$. Thus there is a cycle of length

   $$= P - Q < P - (P - \epsilon) < \epsilon$$

Q.E.D.

Theorem 3 implies that if processor clocks are synchronized to within ϵ, we only need to find cycles of length less than ϵ in a constraint graph to detect a violation. The Floyd-Warshall all-pairs shortest-path algorithm [13] can be used to find the length of the shortest cycles from all vertices in the constraint graph. The complexity of the algorithm is $O(n^3)$, where n is the number of vertices in the graph. Since constraint graphs typically contain a relatively small number of nodes, the algorithm's complexity typically does not constitute a bottleneck.

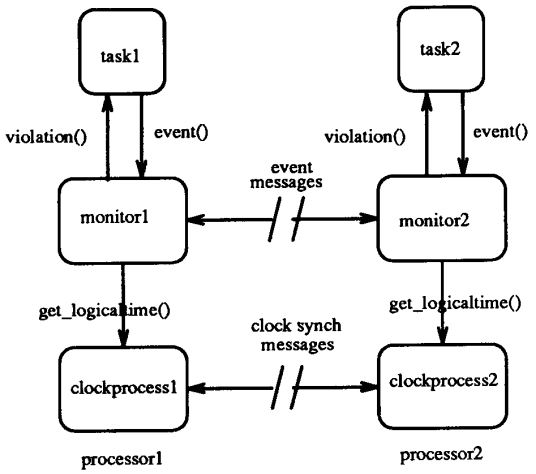

Figure 9: The Layered Interfaces of the Distributed Run-time Monitor

5 A Distributed Run-Time Monitor Prototype

We have implemented a prototype of the distributed monitoring run-time system on a network of IBM RS/6000 workstations connected to a token-ring. The workstations run AIXv.3, a Unix variant that supports the assignment of static priorities to real-time processes, which can immediately preempt other user-processes. Furthermore, a fine-precision timer facility is supported, and it is also possible to differentiate local events that occur 1 μs apart.

The various components of the distributed monitor and their communication links are shown in Figure 9. The clock synchronization layer synchronizes the workstation clocks to within 4 *ms*. The run-time system for the distributed monitor uses the synchronized clocks for timestamping events, and performs several functions. It keeps track of the history of known events, transmits events of interest to other processors, and checks for constraint violations. The application-level processes run on top of this run-time system and are notified when a constraint violation occurs. All communication between components on the same processor, such as event history access and the reading of the synchronized clock, is via shared memory. Message-passing is used only for inter-processor communication. The monitor system, the clock synchronization layer and an X window system based user-interface consist of around 8000 lines of C code. A more detailed description of the prototype, the software interfaces and the user-interface can be found in [11].

5.1 Clock Synchronization Layer

We implemented the probabilistic clock synchronization algorithm described in [3]. Our implementa-

Network Load	Deviation (ms)	Min Interval (secs)	Max Interval (secs)	Max Attempts	Mesgs (avg.)
Normal	2	87	187	6	2.04
Normal	1	36	86	7	2.11
Normal	0.5	8.75	38.75	45	5.41
High	4	53	353	6	3.33
High	1	34	84	64	7.14
High	0.5	3.2	33.2	84	9.09

Table 1: The overhead of achieving synchronization

tion indicates that the message overhead for clock synchronization is clearly acceptable. The message overhead for achieving some clock synchronization bounds is listed in Table 1. All entries assume that the worst case drift rate for local clocks is 10^{-5} and that the probability of loss of synchronization is 10^{-9}. "Normal" in Column 1 refers to a typical load situation on the network where 98% of all round-trip delays take less than or equal to $6ms$. With a 2% probability for an unsuccessful round-trip, a slave which does not receive a response within $6ms$ from the master re-transmits its request a maximum of 6 times. "High" refers to a high-load situation (with file transfers going on in parallel) where a slave waits for $10ms$ before it retries a request. Under these conditions, 60% of all round-trip delays were less than $10ms$, and the distribution of round-trip delays has a significant tail with round-trip delays of as high as $40ms$. Column 2 gives the actual synchronization obtained between the master and a slave. Columns 3 and 4 give the minimum value and maximum value of the interval between successive queries from a slave. Column 5 gives the maximum number of unsuccessful queries by a slave before synchronization is lost. Finally, column 6 gives the average number of messages needed per synchronization, where 2 is the minimum number of messages (one send and one reply).

Every processor runs a clock-slave process that synchronizes its local clock with the clock on a master processor by periodically exchanging messages with the clock-master process.

5.2 Run-Time System Layer

The run-time system for the distributed monitor consists of a set of cooperating monitor processes, one on each processor.

Application tasks inform the local monitor of an event occurrence by putting the event into a queue in shared memory (Figure 10).

If the occurrence time of an event has to be sent to a remote monitor, the monitor puts the event and its local occurrence time into a message and sends it to other monitor processes. Similarly, the monitor receives events from other monitors. If a message arrives from a remote monitor or a timeout occurs, a monitor runs the satisfiability checker using Theorem 3. If a violation is detected, it notifies the application task

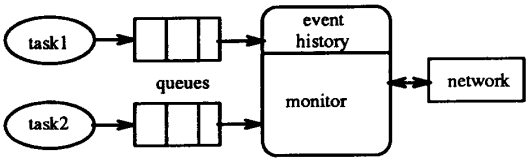

Figure 10: Application Process Interface to its Local Monitor

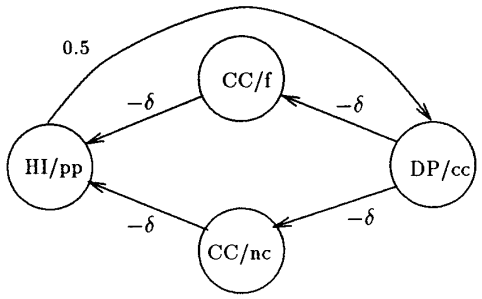

Figure 11: Constraint Subgraph of Tracking System

(with termination as the default action). If there is no violation, a timeout is set based on Lemma 1.

We now illustrate the monitor activities using a subgraph of the constraint graph representing the aircraft tracking system (Figure 11). The monitors on the 3 processors corresponding to HI/pp, CC and DP/cc check the same constraint graph.

When event HI/pp with occurrence index say j happens on processor HI/pp, the corresponding monitor instantiates the vertices of the constraint graph with values from history using index j. Since the local history will have events for HI/pp only[4], the values of @(CC/f, j), @(CC/nc, j) and @(DP/cc, j) will be undefined. Vertices that correspond to events that have not yet occurred are not instantiated. Instead an edge from the unistantiated vertex to the **0** vertex of weight - *current_time* is added (Section 3.4). Next the satisfiability checker based on Theorem 3 is run. The checker flags a violation if there is a cycle of length $< \epsilon$ in a constraint graph. This is pessimistic as some of the constraints in the graph can be between events on a single processor and the other constraints can be between events on different processors. In this case there will not be a violation as the delay and deadline constraints are to non-local events only. Monitor HI/pp sends the time of the j^{th} occurrence of HI/pp to the monitor on processor CC.

When monitor CC receives the message containing the occurrence time of event HI/pp, it stores the

[4]Monitors corresponding to CC and DP/cc do not send their local events to HI/pp's monitor (Section 4.1.1).

occurrence time in its local history. Then it instantiates the vertices of the constraint graph with values from history using index j and runs the satisfiability checker. If events CC/f, CC/nc have not yet occurred, then it sets a timeout at $(0.5 - \delta)$ seconds from @(HI/pp, j). This timeout corresponds to an intermediate deadline on CC/f and CC/nc from HI/pp. If the events CC/f and CC/nc do not occur by the timeout then the constraint has been violated. This is an example of an early detection of a violation. Otherwise, when CC/f and CC/nc occur the constraint graph is checked to ensure that the delay constraint from HI/pp to CC/f and CC/nc is met. Then the occurrence times of CC/f, CC/nc, and their predecessor event, i.e., the j^{th} occurrence of HI/pp, are sent to monitor DP/cc.

6 Conclusions

Run-time monitoring of a distributed real-time system needs to address the issues of constraint specification, clock synchronization, timer granularity, message overhead and time of detection. In this paper we have extended the uniprocessor monitoring model of [2] to a distributed real-time system. The principal advantage of our approach is that derived intermediate constraints can predict the violation of a user-level constraint even before the violation occurs. This can enable the application to take corrective action to adapt to the error condition. We have shown that the problem of minimizing of the number of messages exchanged between processors while still detecting violations at the earliest possible time is intractable. However, for one common class of constraints which arises whenever processing occurs in pipelined stages, message-passing requirements can be readily minimized. The drift among the various processor clocks can also be taken into account with clock synchronization.

This work can be extended in several directions. A major concern in real-time systems is the need to quantify the intrusiveness of the monitoring activities on the timing behavior of the real-time application. Hence, monitoring activities must themselves be scheduled and included in a scheduling analysis of the system [8]. A higher-level programming interface is needed to specify the monitored constraints and their communication requirements. Executable specifications of real-time systems that automatically generate run-time monitoring code are also of interest.

Acknowledgements

We would like to thank Alan Shaw for his comments on an earlier draft of this paper.

References

[1] K. Arvind. A New Probabilistic Algorithm for Clock Synchronization, *Proc. IEEE Real-Time Systems Symp.*, pp. 330-339, Dec. 1989.

[2] S. Chodrow, F. Jahanian, and M. Donner. Run-Time Monitoring of Real-Time Systems, *Proc. IEEE Real-Time Systems Symp.*, pp. 74-83, Dec. 1991.

[3] F. Cristian. Probabilistic Clock Synchronization, *Distributed Computing 3*, pp. 146-158, 1989.

[4] M. R. Garey and D. S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*, W. H. Freeman and Company, 1979.

[5] D. Haban and K. G. Shin. Application of Real-Time Monitoring to Scheduling Tasks with Random Execution Times, *Proc. IEEE Real-Time Systems Symp.*, pp. 172-181, Dec. 1989.

[6] D. Haban and D. Wybranietz. A Hybrid Monitor for Behavior and Performance Analysis of Distributed Systems. *IEEE Trans. on Software Eng. 16,2*, pp. 197-211, Feb. 1990.

[7] F. Jahanian and A. Goyal. A Formalism for Monitoring Real-time Constraints at Run-time, *Proc. IEEE Fault-Tolerant Computing Symp.*, pp. 148-155, June 1990.

[8] F. Jahanian and R. Rajkumar. An Integrated Approach to Monitoring and Scheduling in Real-Time Systems, *IEEE Workshop on Real-Time Operating Systems and Software*, May 1991.

[9] C. Kilpatrick, K. Schwan and D. Ogle. Using Languages for Capture, Analysis and Display of Performance Information for Parallel and Distributed Applications, *International Conf. on Computer Languages*, March 1990.

[10] J. Lundelius and N. Lynch. An Upper and Lower Bound for Clock Synchronization, *Information and Control 62*, pp. 190-204, 1984.

[11] S. Raju, R. Rajkumar and F. Jahanian. Monitoring Timing Constraints in Distributed Real-time Systems, *TR-92-09-03*, Dept. of Computer Science and Eng., University of Washington, Sept. 1991.

[12] R. Snodgrass. A Relational Approach to Monitoring Complex Systems. *ACM Trans. on Computer Systems 6,2*, pp. 157-196, May 1988.

[13] R. E. Tarjan. *Data Structures and Network Algorithms*, Society for Industrial and Applied Mathematics, 1983.

[14] H. Tokuda, M. Koreta and C.W. Mercer. A Real-Time Monitor for a Distributed Real-Time Operating System, *ACM Sigplan Notices 24,1*, pp. 68-77, Jan. 1989.

[15] J. P. Tsai, K-Y Fang and H-Y Chen. A Non-invasive Architecture to Monitor Real-time Distributed Systems, *IEEE Computer 23,3*, pp. 11-23, March 1990.